

普通高等教育“十一五”国家级规划教材



21世纪大学本科计算机专业系列教材

朱庆生 古平 主编
李晓明 主审

Java程序设计（第2版）

http://www.tup.com.cn

- 国家精品课程教材
- 根据教育部“高等学校计算机科学与技术专业规范”组织编写
- 与美国ACM和IEEE CS *Computing Curricula* 最新进展同步

清华大学出版社

普通高等教育"十一五"国家级规划教材
21 世纪大学本科计算机专业系列教材

Java 程序设计
（第 2 版）

朱庆生　古　平　主编
刘　骥　葛　亮　杨瑞龙　编著

清华大学出版社
北　京

内 容 简 介

本书在内容组织和编写中充分融入了作者在承担"Java程序设计"国家精品课程和国家精品资源共享课程建设和教改中的成果，更加契合高等学校课程教学模式和学生渐进式学习的特点。遵循学生学习的认知过程和规律，以"兴趣为先、任务驱动、学以致用"的理念为指导思想，每个章节编写均采用WWH教学方式，即通过"理解知识概念(What)、把握原理和规律(Why)、掌握实现技巧(How)"，有机地将知识、理论融入到编程实践中。

本书在知识表达形式上改变了传统教材对知识平铺直叙的描述方式，而是将大量知识点融入到代码实例中进行讲解，由浅入深，通过对核心知识的实例化来提高学生的编程兴趣，让学生既知道需要学什么，又懂得如何学以致用。与其他Java教材相比，本书在重要知识点和案例选择中均针对实际问题需求，从软件开发者的视角进行讲解与分析，各知识点中均凝聚了作者在Java应用开发中的实际经验和教研成果，体现了知识的实用性。在注重基本知识的同时，将案例解决方法、过程和原理融入其中，提升学生分析问题和实际编程的能力。

本书可作为高等学校计算机及相关专业本科生"Java程序设计"课程的教材，也可作为需要运用Java语言开发应用的软件人员的自学参考书。

图书在版编目(CIP)数据

Java程序设计/朱庆生，古平主编. —2版. —北京：清华大学出版社，2017(2018.2重印)
(21世纪大学本科计算机专业系列教材)
ISBN 978-7-302-45478-6

Ⅰ. ①J… Ⅱ. ①朱… ②古… Ⅲ. ①JAVA语言－程序设计－高等学校－教材 Ⅳ. ①TP312.8

中国版本图书馆CIP数据核字(2016)第275235号

责任编辑：张瑞庆
封面设计：何凤霞
责任校对：梁 毅
责任印制：刘祎淼

出版发行：清华大学出版社
网 址：http://www.tup.com.cn，http://www.wqbook.com
地 址：北京清华大学学研大厦A座 **邮 编**：100084
社 总 机：010-62770175 **邮 购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn，010-62795954
印 刷 者：北京富博印刷有限公司
装 订 者：北京市密云县京文制本装订厂
经 销：全国新华书店
开 本：185mm×260mm **印 张**：26.75 **字 数**：633千字
版 次：2011年5月第1版 2017年1月第2版 **印 次**：2018年2月第4次印刷
印 数：21001～23000
定 价：49.00元

产品编号：070982-01

本书主审：李晓明

第 2 版前言

FOREWORD

Java 语言是由 Sun 公司推出的面向对象程序设计语言，它的平台无关性特点非常适于网络应用，因而备受青睐。由于 Java 的体系和内容非常庞大，如何帮助初学者建立一个完整的知识体系，如何引导读者将 Java 的理论、实践和编程技能融为一体，是本书重点考虑的问题。

本书是网络教育国家精品课程和国家精品资源共享课程“Java 程序设计”的配套教程，适合作为高等学校或网络教育 Java 课程的教材。

本书第 1 版出版 5 年多来，受到全国广大师生的好评，已 7 次印刷发行。教材第 2 版根据教育部高等学校计算机类专业教学指导委员会关于提升学生编程实践能力的精神，在总结课程教学改革经验基础上并结合对第 1 版教材的反馈意见修订而成。全书组织架构与第 1 版保持不变，内容由浅入深地分为 4 篇：“基础篇”（第 1～6 章）介绍面向对象程序设计的基本思想和 Java 核心语法；“提高篇”（第 7～10 章）讲述如何利用 Java 编程接口提升程序的实用性、可靠性和界面友好性；“网络篇”（第 11～14 章）在介绍网络通信知识基础上，讲述如何应用 Java 语言实现桌面应用和 Web 应用；“实例篇”（第 15 章）通过具体案例展示 Java 应用程序的设计过程、编程思路和技术，为项目实战积累编程经验。

第 2 版主要修改包括：

（1）根据 JDK 8 版的新特性进行了版本更新，并增加了若干相关知识点，具体内容有：第 1 章增加了 Eclipse 安装使用，同时删除原附录 A；第 3 章增加了数组类 Arrays；第 5 章增加了内部类、泛型、闭包、反射机制和注解技术等；第 6 章增加了正则表达式、Runtime 类、自动装箱和拆箱技术等；第 9 章增加了线程池、多线程互斥与同步等；第 10 章弱化 Applet，同时增强了 Swing 中的高级组件；第 11 章增加了多播通信技术；第 14 章增加了数据库事务管理等内容。

（2）进一步丰富了导入案例、知识点案例和各章的综合案例，增加了一系列面向小节知识的实用小案例，力图通过解决模拟问题，培养学生思维能力，激发学生编程兴趣。

（3）提供了与本书配套的、完整的案例源码，便于学生课后自我练习。

全书由朱庆生、古平、刘骥、葛亮、杨瑞龙编写。其中，朱庆生教授负责全书体系构思、编写框架、内容选择和编写指导等，并对全书进行审定和修改。第 3、4、5、11 章由古平副教授编写，第 1、7、12、14 章由葛亮副教授编写，第 10、13、15 章由刘骥老师编写，第 2、6、8、9 章由杨瑞龙老师编写。

本次修订，错误或遗漏之处在所难免，恳请批评指正。

作　者

2016 年 9 月

第1版前言

FOREWORD

Java 语言是由 Sun 公司推出的面向对象程序设计语言，它的平台无关性特点非常适于网络应用，因而备受青睐。Java 语言正逐步成为全世界程序员使用最多的编程语言，并在桌面应用、Web 应用、分布式应用、嵌入式应用系统中发挥着重要作用。

对于初学 Java 程序设计语言的人来说，Java 的体系和内容显得非常复杂。如何帮助他们建立一个完整的知识体系，以及如何引导他们将 Java 的理论、实践和技能融为一体，是本书重点考虑的问题。

本书是在国家精品课程"Java 程序设计"(网络)和教育部-IBM 精品课程"Java 并发编程实践"的建设和实践教学中总结出来的实用教材，面向全国高校计算机及其相关专业本科生，也可供 Java 自学人员、软件开发人员参考使用。

全书内容由浅入深分为 4 篇。第一篇为基础篇(第 1～6 章)，侧重介绍面向对象程序设计的基本思想和 Java 核心语法，包括 Java 运行环境、程序设计基础、数组、类和对象、继承与多态、Java 标准类库等。通过这一部分的学习，读者能够掌握 Java 程序设计的基本方法，并能设计简单的 Java 程序。

第二篇为提高篇(第 7～10 章)，讲述如何利用 Java 开放的编程接口提升程序的实用性、可靠性和界面友好性，包括 Java 的异常处理机制、输入输出处理、多线程、图形界面设计、Applet 等。通过这一部分的学习，读者可以开发功能更复杂、界面更友好的实用 Java 应用系统。

第三篇为网络篇(第 11～14 章)，介绍网络通信的基础知识，以及 Java 对桌面网络应用和 Web 应用的支持与实现，内容包括 URL 与 Socket 通信技术、Web 编程技术、JDBC 技术，以及上述技术在 Tomcat 与 Eclipse 开发平台中的具体实现。这是 Java 在网络通信、Web 服务中应用的关键技术。

第四篇为实例篇(第 15 章)，通过两个实用案例，分别展示 Java 桌面应用和 Web 应用程序的设计过程、思路和关键技术，是前 3 篇知识的综合应用，也为项目实战积累开发经验。

本书在编写上具有以下特点：

(1) 注重知识点之间的相互衔接。本书在章节安排上考虑了知识点之间的依赖关系，分析时层层推进，注重知识点之间的环环相扣。

(2) 采用 Why—What—How 的编写模式。本书采用"为什么学→学什么→如何学"的教学思路进行内容组织，每章首先通过"本章学习目标"让读者明确为什么学，其次通过具体案例引出需要学什么，然后从问题求解过程中逐步引出关键知识点，使读者知道该如何学，

能够轻松、快速掌握 Java 语言的基本知识体系和编程技巧。

(3) 案例层次化。在章节进入时有导入案例,主要知识点有印证案例,模拟练习时有实训案例,所有案例均配有详细的代码解释和分析。导入案例→印证案例→实训案例,实践→理论→再实践,理论知识与实际应用紧密结合。

本书由朱庆生、古平、刘骥、葛亮、杨瑞龙编写。其中,朱庆生教授负责全书体系构思、框架制定、内容选择、编写指导等,并对全书各章进行了审查和修改;第 3、4、5、10(部分)、11 章由古平编写;第 1、7、12、14 章由葛亮编写;第 10(部分)、13、15 章和附录由刘骥编写;第 2、6、8、9 章由杨瑞龙编写。

书中难免有遗漏和不足之处,恳请业界同仁及读者朋友提出宝贵意见,以便在修订时进一步完善。

编　者

2011 年 1 月

目　录

CONTENTS

第一篇　基　础　篇

第二篇　提　高　篇

第三篇 网 络 篇

第四篇　实　例　篇

第一篇　基 础 篇

第 1 章 Java 概述

本章学习目标

Java 是目前最流行的编程语言，它以其强大的移植能力、多线程处理和网络处理能力成为研究人员和开发人员瞩目的焦点。本章将开启 Java 的学习之旅，除了解为什么要使用 Java，以及如何使用 Java 外，还需要掌握以下内容：

（1）Java 语言的特点。

（2）Java 开发运行环境的配置。

（3）Java 程序的种类。

（4）简单 Java 程序的开发。

1.1 认识 Java 语言

Java 语言是 20 世纪 90 年代由 Sun Microsystems 公司开发的革命性的编程语言，它被美国著名的专业杂志 PC Magazine 评为 1995 年十大优秀科技产品之一。之所以称 Java 为革命性的编程语言，是因为传统的软件往往与具体的实现环境有关，一旦环境有所变化就需要对软件做一番改动，既费时又耗力；而用 Java 语言编写的软件能在执行码的层次上兼容，只要计算机提供了 Java 虚拟机环境，用 Java 语言编写的软件就能在其上运行。

Java 技术的通用性、高效性、平台移植性和安全性，使之成为网络计算的理想技术。从笔记本电脑到数据中心，从游戏控制台到科学超级计算机，从手机到互联网，Java 无处不在！它在各个重要的行业部门得到了广泛的应用，而且出现在各种各样的设备、计算机和网络中。目前，采用 Java 的设备包括 97%的企业桌面、美国 89%的桌面（或计算机）、30 亿部移动电话、100%的蓝光盘播放器、50 亿张 Java 卡、1.25 亿台 TV 设备等（来源：www.java.com）。

迄今为止，Java 平台已吸引了 900 万软件开发者。这是全球最大的、最具活力的开发团队。凭借其卓越的通用性、高效性和移植性，Java 对开发者具有不可估量的价值，使他们可以完成以下工作：

（1）在一个平台上编写软件，然后即可在几乎所有其他平台上运行。

（2）创建可在 Web 浏览器和 Web 服务中运行的程序。

(3) 开发适用于在线论坛、存储、投票、HTML 格式处理以及其他用途的服务器端应用程序。

(4) 将采用 Java 语言的应用程序或服务组合在一起,形成高度定制的应用程序或服务。

(5) 为移动电话、远程处理器、低成本的消费产品以及其他任何具有数字核心的设备编写强大而高效的应用程序。

Java 已经成为最流行的编程语言。

1.1.1 Java 语言的特点

Java 语言是一种高级编程语言,它具有简单、结构中立、面向对象、可移植、分布式、高性能、多线程、健壮、动态和安全等特点。

在用 Java 语言进行程序开发时,首先以纯文本的方式编写所有的 Java 源程序,并保存成以.java 为后缀名的文件;然后将这些源程序用 javac 编译器编译成.class 后缀名的字节代码文件;字节代码不是被本地处理器执行的代码,而是能够被 Java 虚拟机(Java Virtual Machine1,Java VM)执行的代码。最后用 Java 运行工具在 Java 虚拟机中执行 Java 应用程序。Java 程序的开发过程如图 1-1 所示。

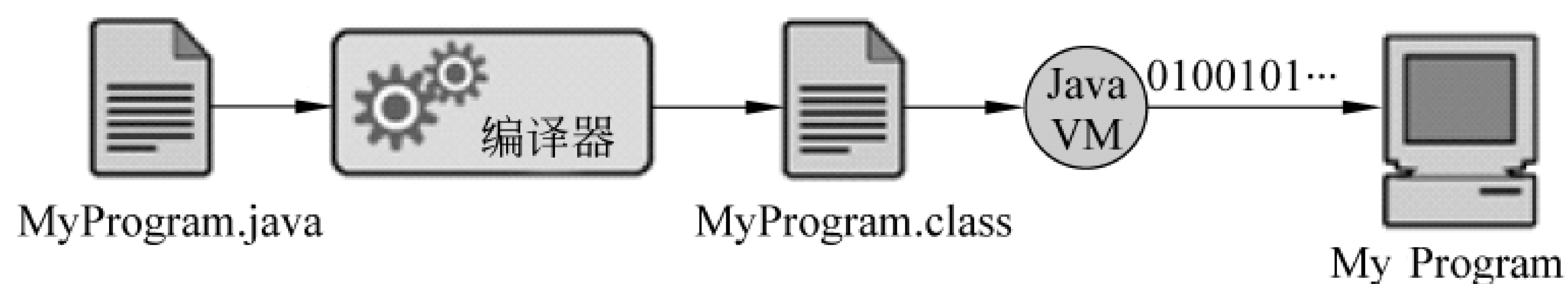


图 1-1 Java 程序的开发过程

由于 Java 虚拟机可以运行在不同的操作系统之上,因此同一个字节代码文件可以在 Windows、Solaris OS、Linux、UNIX 和 Mac OS 等操作系统上运行,如图 1-2 所示。

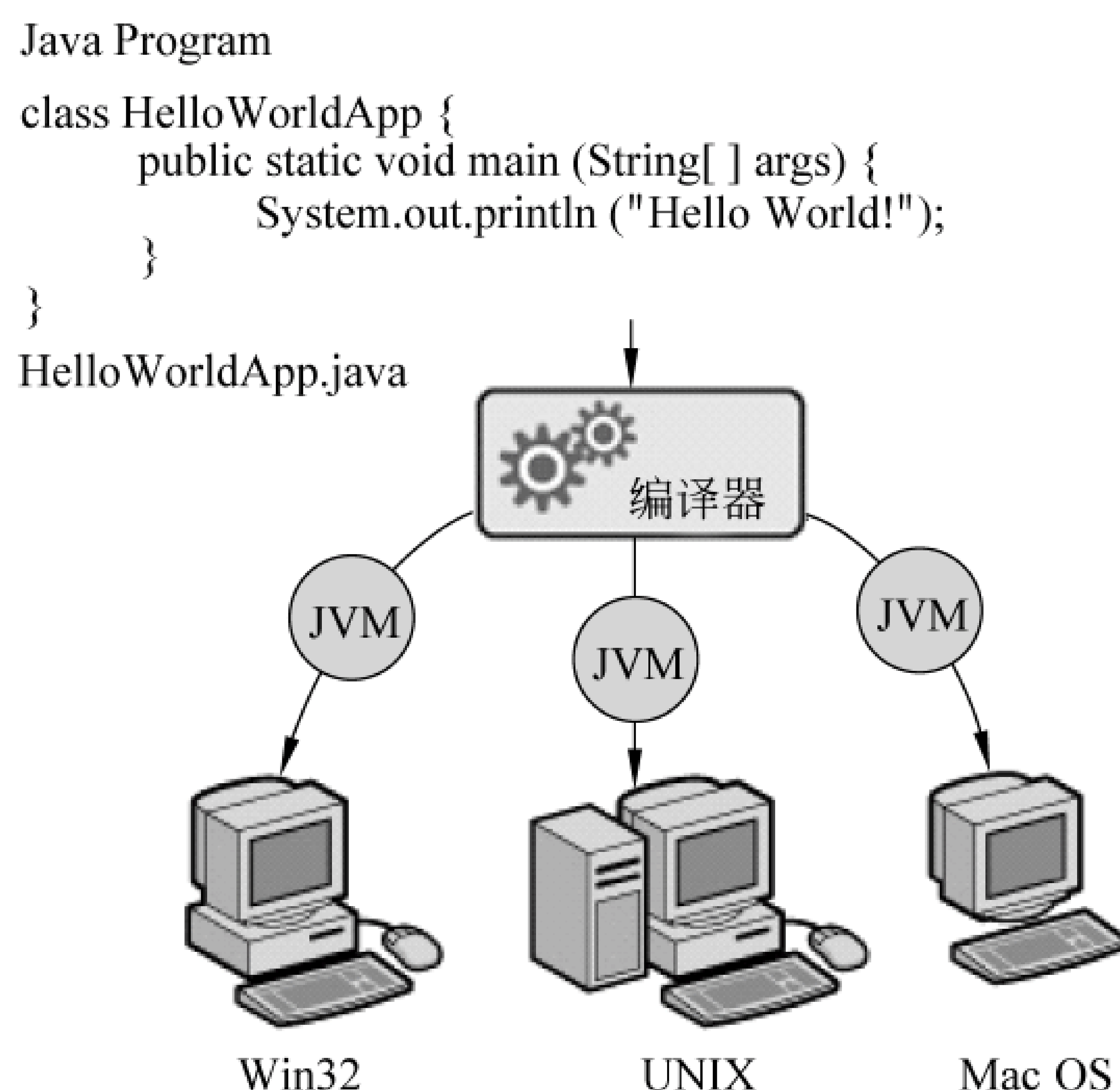


图 1-2 同一个 Java 程序可以通过 Java 虚拟机在各种平台上运行

1.1.2 Java 平台的体系结构

Java 不仅是编程语言,还是一个强大的软件平台。Java 技术为程序员的软件开发提供

了多种支持，主要包括如下方面。

1. 开发工具

Java 平台提供了各种开发工具，包括编译器、解释器、文档生成器以及文件打包工具等。一位初级程序员主要使用的工具是编译器 javac、解释器 java 以及文档生成器 javadoc。

2. 应用程序编程接口

应用程序编程接口（Application Programming Interface，API）包含了对软件常用功能的支持，涉及基本数据对象、用户界面、网络与安全、XML 生成和数据库访问等众多方面。这些 API 主要可以分为三大类：

（1）Java 核心 API，由 Sun 公司制定的基本 API，任何 Java 平台都必须提供。

（2）Java 标准扩展 API(javax)，由 Sun 公司制定的扩充 API，Java 平台可以选择性地提供或加载。

（3）厂商或者组织所提供的 API，由各家公司或组织所提供。

Java API 的制定过程是公开的，并且有许多业界技术领先的公司共同参与，所以相当完善而优异。与 Java 标准相关的资料，都可以在 http://www.oracle.com/technetwork/java/index.html 获得。

目前 Java 的体系结构已经变得相当庞大。Sun 公司在 1998 年将 Java 平台划分成 J2EE、J2SE 和 J2ME 共 3 个版本（目前改称为 Java EE、Java SE 和 Java ME），针对不同的市场目标和设备进行了定位。Java EE(Java Platform，Enterprise Edition)的主要目的是为企业计算提供一个应用服务器的运行和开发平台。Java EE 本身是一个开放的标准，任何软件厂商都可以推出自己的符合 Java EE 标准的产品，使用户可以有多种选择。IBM、Oracle、BEA 和 HP 等多家公司已经推出了自己的产品。例如，BEA 公司的 WebLogic、IBM 公司的 WebSphere 等。Java SE(Java Platform，Standard Edition)的主要目的是为台式机和工作站提供一个开发和运行平台。Java SE 是 Java 技术的基础，在学习 Java 的过程中，将首先学习 Java SE。Java ME(Java Platform，Micro Edition)主要面向电子消费产品，目的是为电子消费产品提供一个 Java 的运行平台，使得 Java 程序可以在手机、PDA 和机顶盒等产品上运行。由于技术的发展，Java ME 已经很少使用，但近几年来占市场主流的智能手机系统——Android，使用 Java 作为开发语言。因此，Java 也是移动开发领域最常用的语言。

1.2 Java 运行环境与开发环境

Java 提供了一个免费的 Java 开发工具集（Java Development Kits，JDK），编程人员可以利用这些工具来开发或者调试 Java 程序。通常以 JDK 的版本来定义 Java 的版本。JDK 1.0 版于 1996 年 1 月公开，目前的最新版是 JDK 8 Update 91。

JDK 简单易学，是开发和运行 Java 程序的基本方法。但 JDK 不是一个集成开发环境，不方便进行复杂的 Java 软件开发，也不利于团体协作开发。目前主流的 Java 开发方法都是通过 Eclipse、NetBeans、JBuilder 等集成开发环境进行的。本书将采用 Eclipse 进行 Java 程序的开发和调试，有关 Eclipse 的安装和基本使用方法参见本书 1.2.3 节中的介绍。

1.2.1 Java 运行环境

如果只想运行已有的 Java 程序，可以只安装 Java 运行环境(Java Runtime Environment,JRE),JRE 由 Java 虚拟机、Java 的核心类以及一些支持文件组成。可以在网站 http://www.java.com/上免费下载 Java 的 JRE,下载时选择适当的操作系统(如图 1-3 所示),然后根据提示进行安装即可。

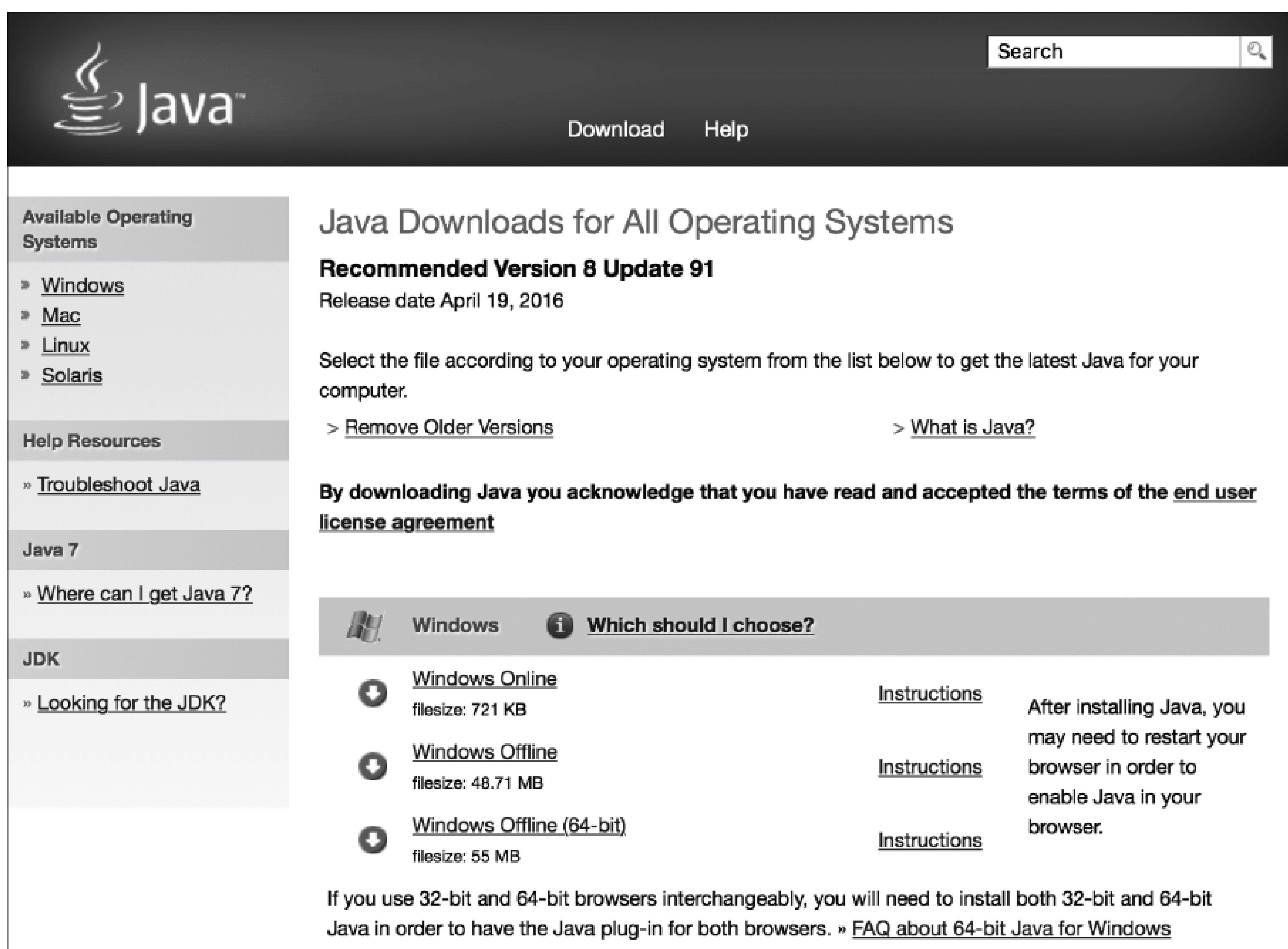


图 1-3 下载 JRE 时选择适当的操作系统

1.2.2 安装

为了开发 Java 应用程序，需要安装 JDK(Java Development Kits)。在安装 JDK 的同时将安装 JRE。在网站 http://www.oracle.com/上可以免费下载 Java 的 JDK。根据提示可以将支持 64 位 Windows 操作系统的 JDK 安装程序 jdk-8u91-windows-x64.exe 下载到本地硬盘，然后根据提示进行安装即可。成功安装后，在 JDK 目录下有 bin、lib、jre、demo 等子目录，如图 1-4 所示。其中，bin 目录存放了 javac、java 等命令程序，lib 目录存放了 Java 的类库文件，jre 目录是 Java 的运行时环境(JRE),demo 目录保存了许多 Java 示例程序。

1.2.3 安装 Eclipse 开发工具

Eclipse 是一款开放源代码的、跨平台的、基于 Java 的可扩展集成开发环境(IDE)。它最初由 IBM 公司开发，在 2001 年 11 月贡献给开源社区，现在它由非营利软件供应商联盟 Eclipse 基金会(Eclipse Foundation)管理。目前 Eclipse 已经发布了代号为 Mars.2 的 4.5.2 版。时至今日，Eclipse 已经成为最主要的 Java 开发工具。但 Eclipse 的用途并不局限于 Java 领域。Eclipse 提供了一种称为插件(Plug-in)的扩展机制，允许软件开发人员构建与

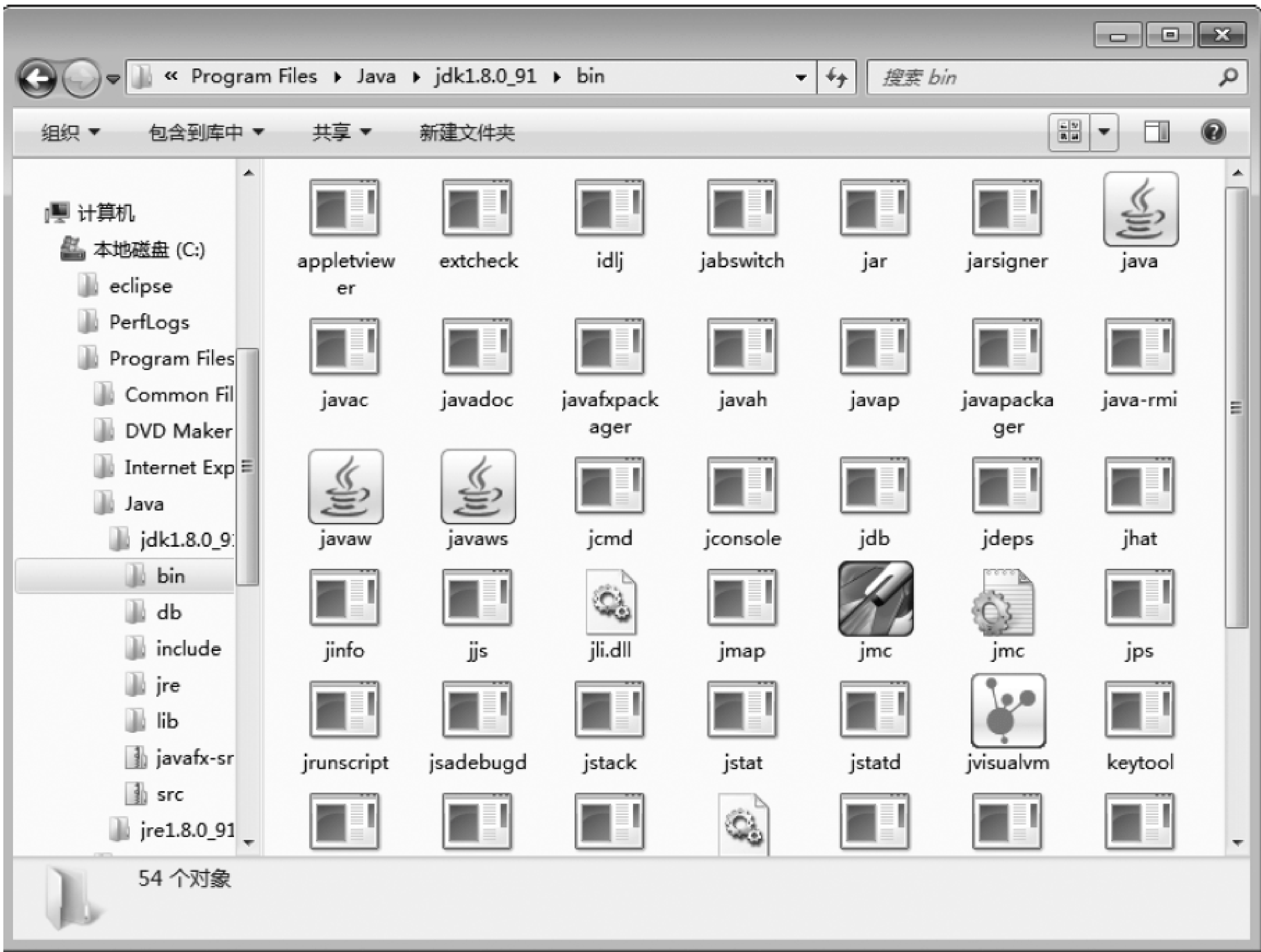


图 1-4　JDK 的目录结构及文件

Eclipse 环境无缝集成的插件。通过插件可以让 Eclipse 具有更多的功能。这使得 Eclipse 不仅可以作为 Java 的开发工具,还可以作为 C/C++、Ruby、Python 等语言的开发工具。

要使用 Eclipse,首先需要安装 JDK 环境(见本书 1.2.2 节),然后从 Eclipse 官方网站(http://www.eclipse.org/downloads/,如图 1-5 所示)选择 Eclipse IDE for Java Developers 和

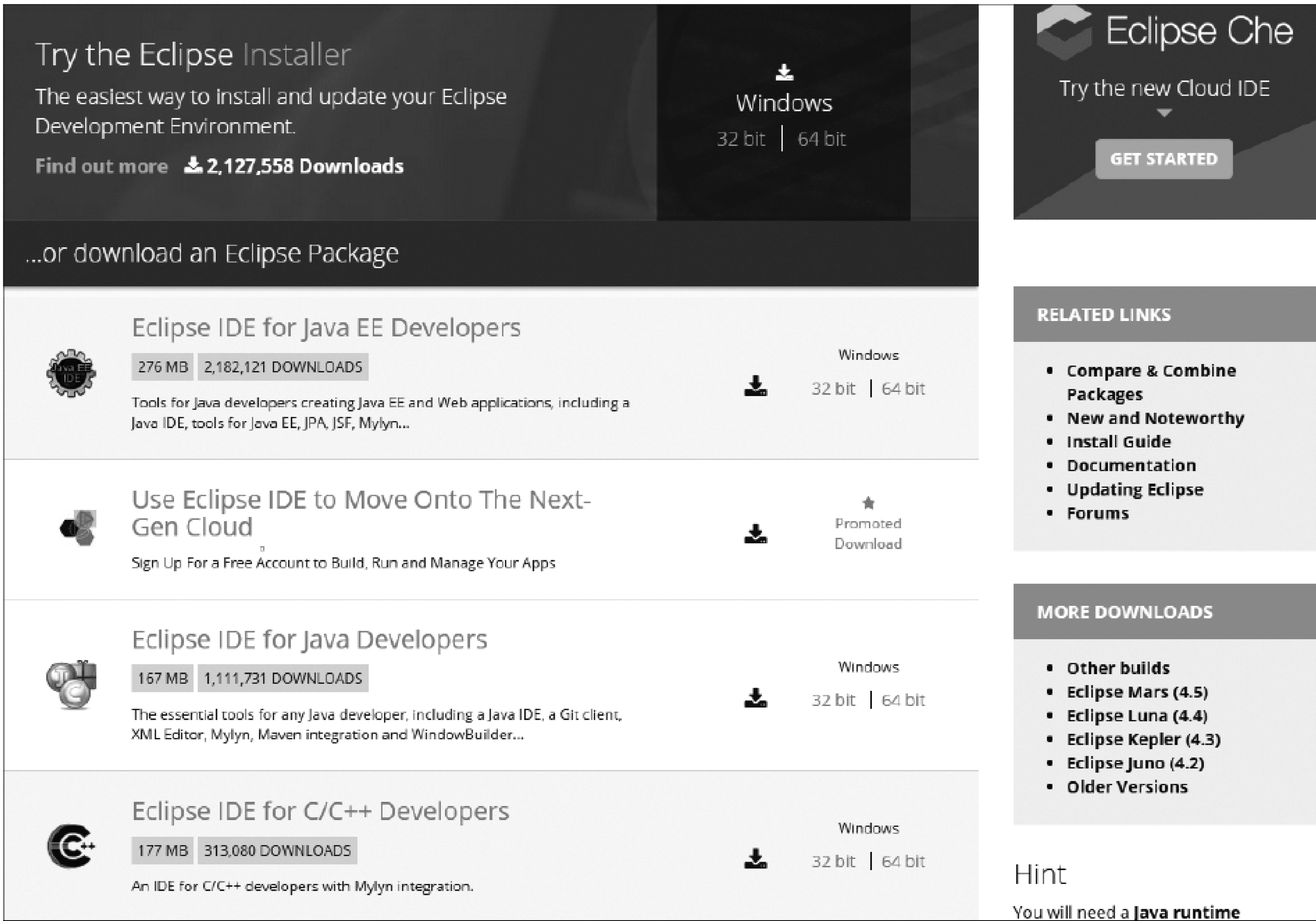


图 1-5　下载 Eclipse

对应的操作系统进行下载。本书选择下载 64 位 Windows 版本,下载的文件为 eclipse-java-mars-2-win32-x86_64.zip。下载完成之后,解压下载的文件到任意目录即可完成安装。

1.3　Java 程序举例

常见的 Java 程序主要有两类:Application(应用程序)和 Servlet(服务器端小程序)。应用程序在计算机中单独运行,而 Servlet 是运行在服务器端的小程序,它可以处理客户端传来的请求(request),然后将处理结果以响应(response)的方式传回给客户端。本节将介绍简单的 Java 应用程序,关于服务器端小程序的内容将在第 12 章中介绍。本节中的例子将在 Eclipse 中进行开发。

1.3.1　用 Eclipse 编写第一个 Java Application

【例 1.1】　用 Eclipse 编写 Java Application,要求在命令行窗口中显示"Hello, world!"。

步骤如下。

(1) 运行 Eclipse。

首先在 Eclipse 的安装目录上双击 eclipse.exe,以运行 Eclipse(如图 1-6 所示)。

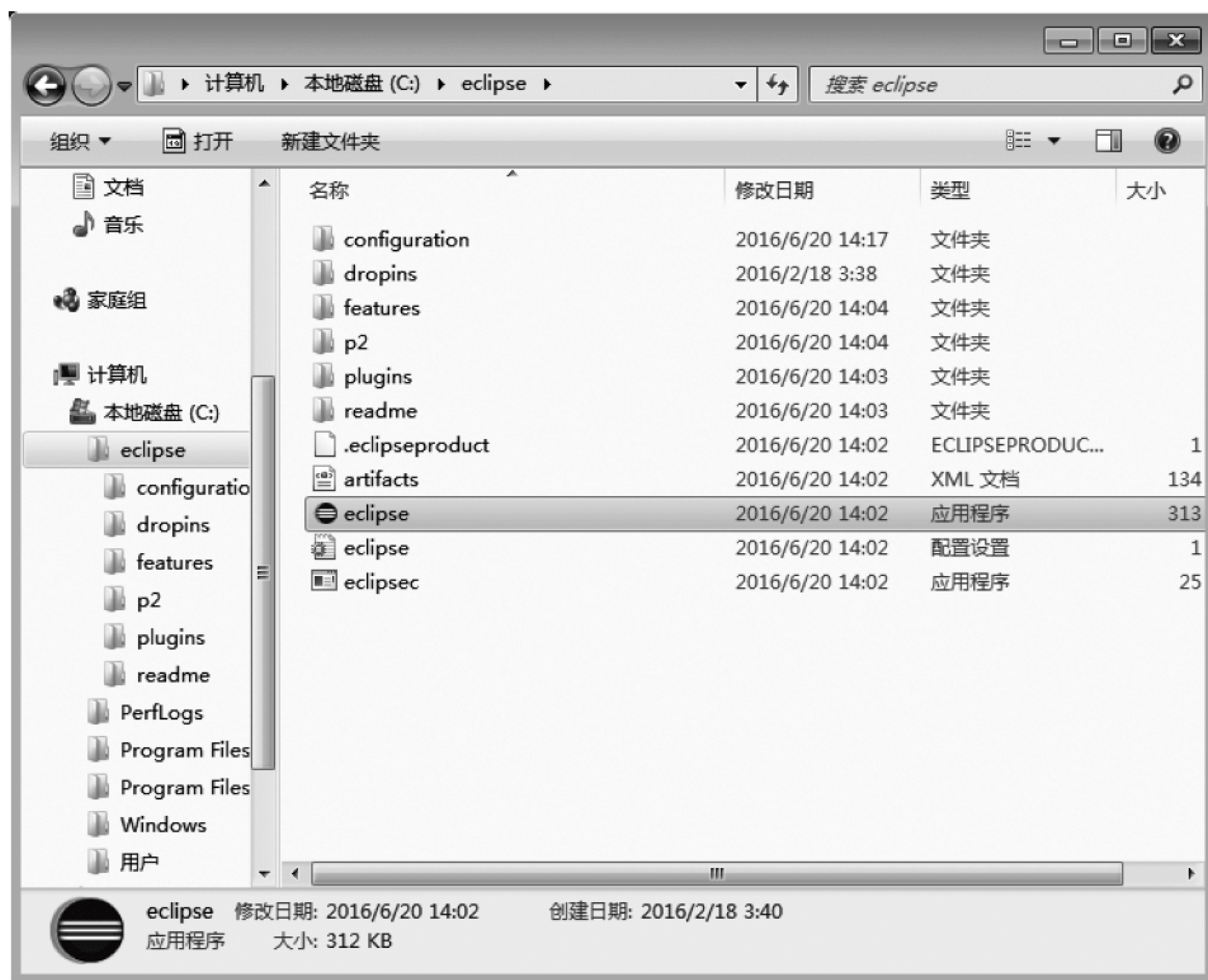


图 1-6　Eclipse 的安装目录

首次运行 Eclipse 会出现如图 1-7 所示的界面。该界面提示用户为 Eclipse 选择一个工作空间(workspace)。所谓工作空间,是 Eclipse 存放源代码的目录。本书选择 C:\

workspace 作为工作空间，今后创建的 Java 源程序就存放在该目录。勾选 Use this as the default and do not ask again 选项，则今后使用 Eclipse 时不会再弹出该对话框。

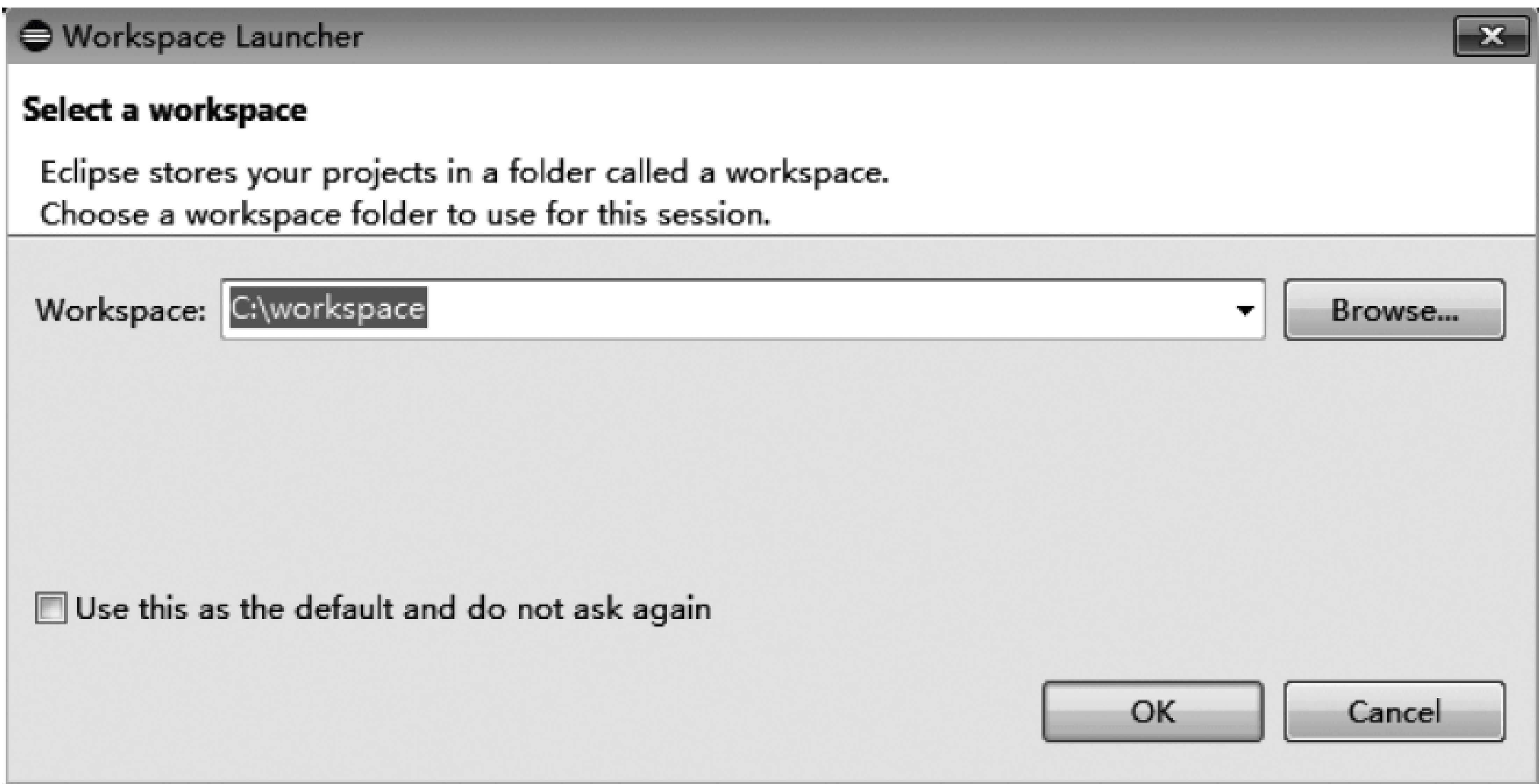


图 1-7　选择工作空间

Eclipse 运行后的界面如图 1-8 所示。在界面左侧列出了当前工作空间里已有的 Java 工程。界面中部是 Java 源代码的编辑窗口。界面右侧是函数大纲，列出了当前编辑的 Java 代码的所有函数。界面下方是错误等信息提示。

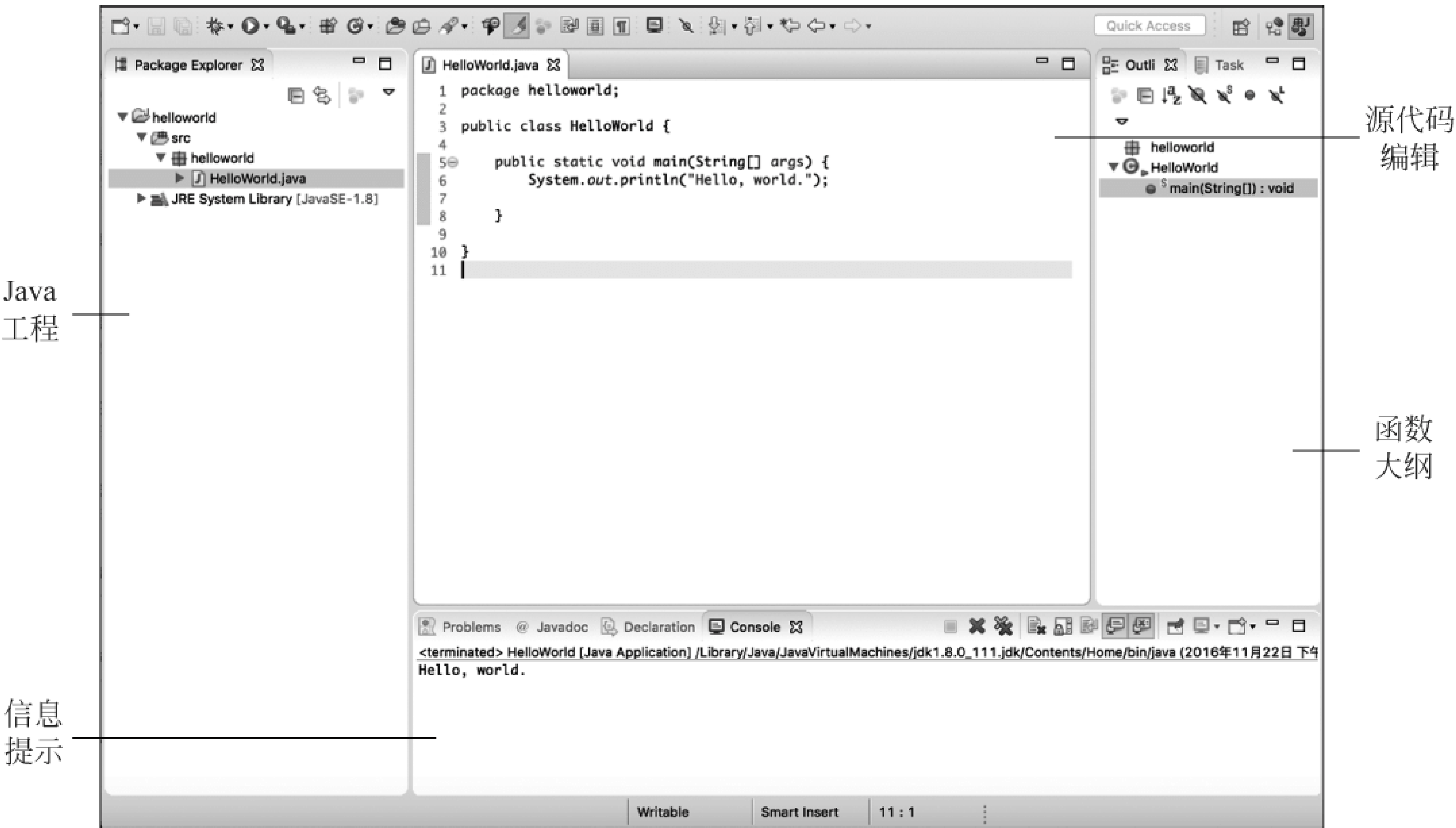


图 1-8　Eclipse 的运行界面

(2) 新建 Java 工程。

若要在 Eclipse 中编写 Java 代码，必须首先新建一个 Java 工程(Java Project)。选择菜单项 File，然后选择 New→Java Project 选项，就会弹出新建 Java 工程的对话框(如图 1-9 所示)。在新建 Java 工程对话框中输入 Java 工程的名称(如 code01)，单击 Finish 按钮，就会在工作空间中新建一个 Java 工程，如图 1-10 所示。

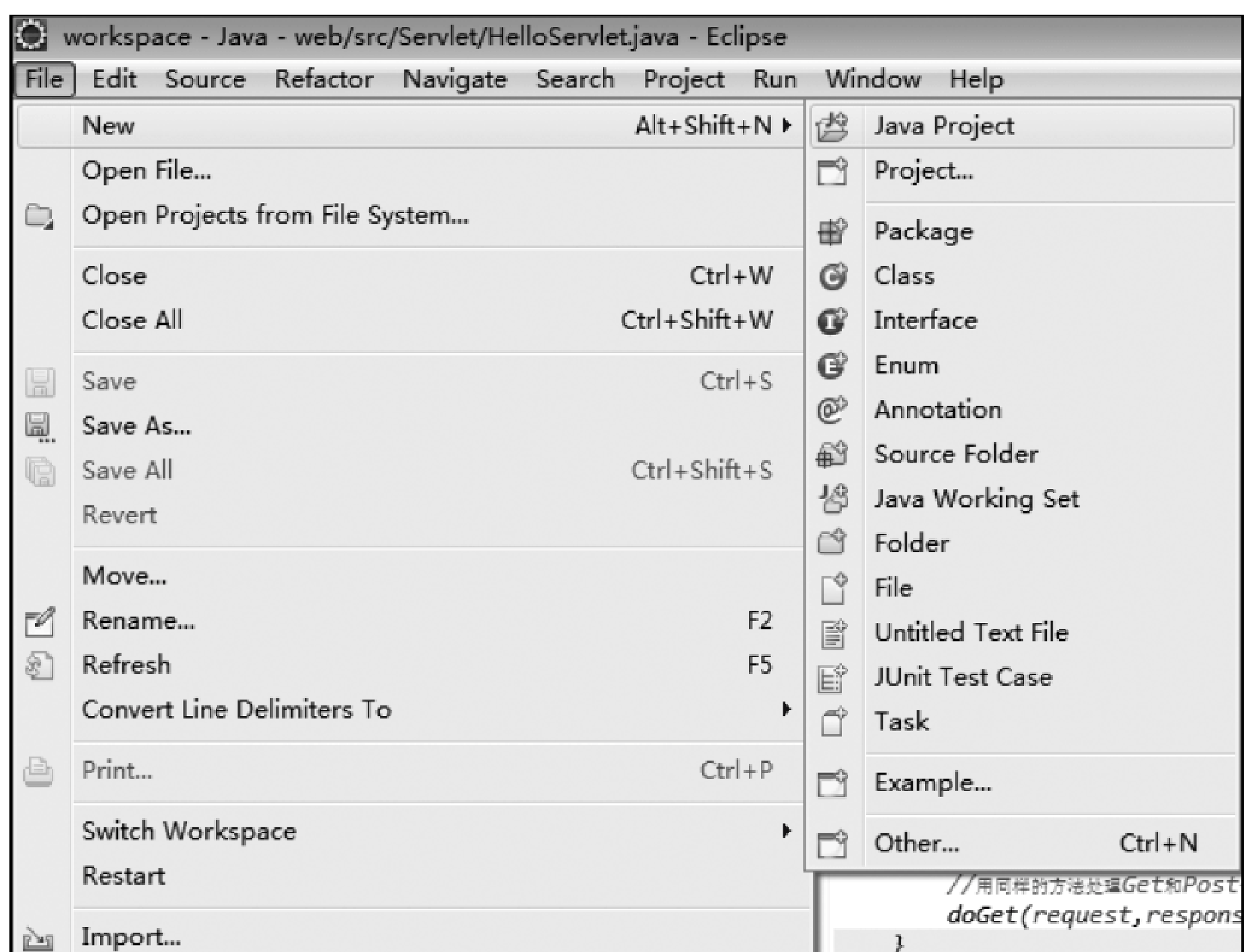


图 1-9　通过菜单新建 Java 工程

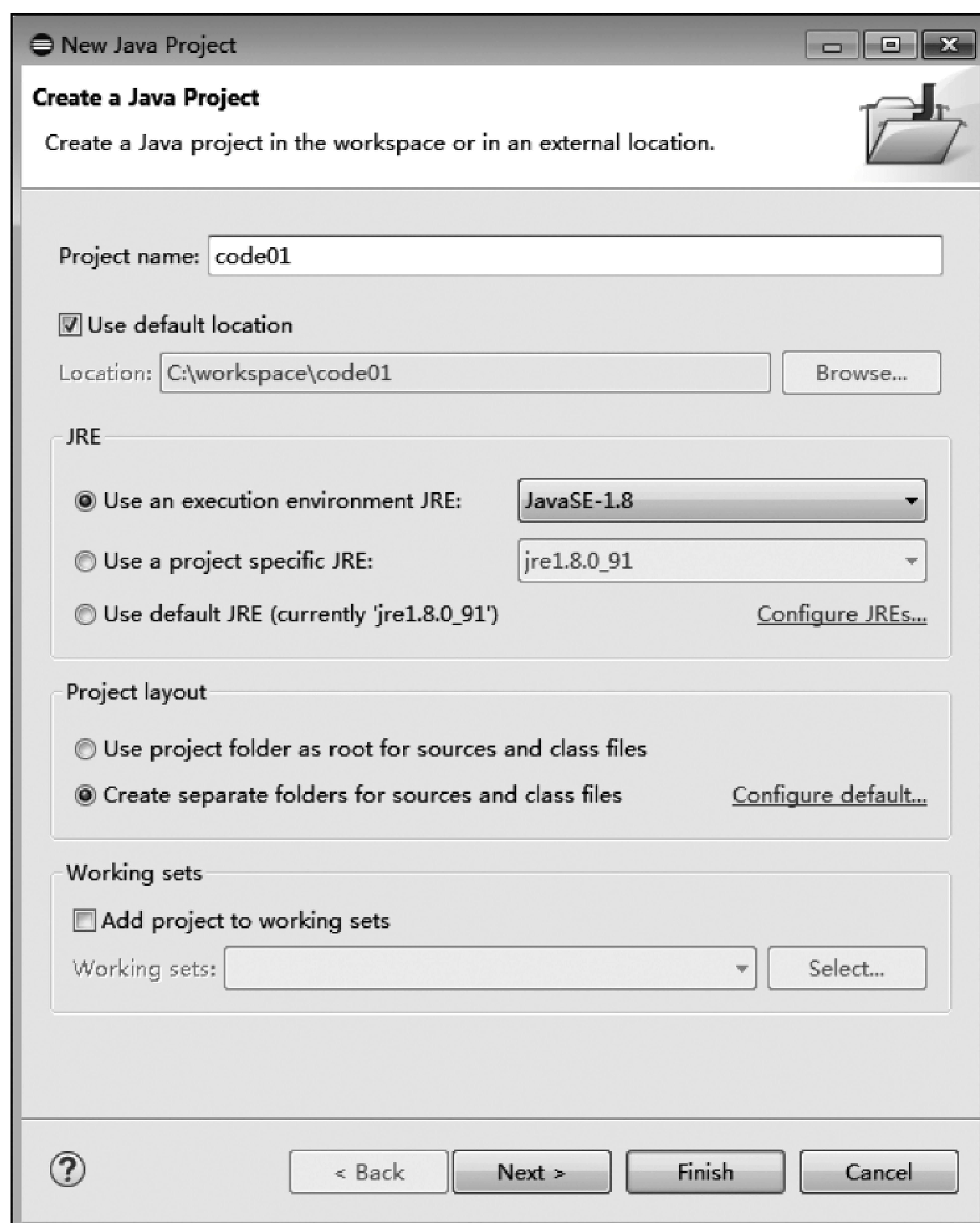


图 1-10　新建 Java 工程对话框

（3）新建 Java 类。

在 Eclipse 中选中 code01 下的 src 目录（src 目录用于存放 Java 程序的源代码），打开右键菜单，选择 New→Class 选项，新建一个 Java 类（如图 1-11 所示），弹出如图 1-12 所示的对话框。在图 1-12 对话框中输入新建 Java 类的包名、类名等信息，然后单击 Finish 按钮就可以完成 Java 类的新建。

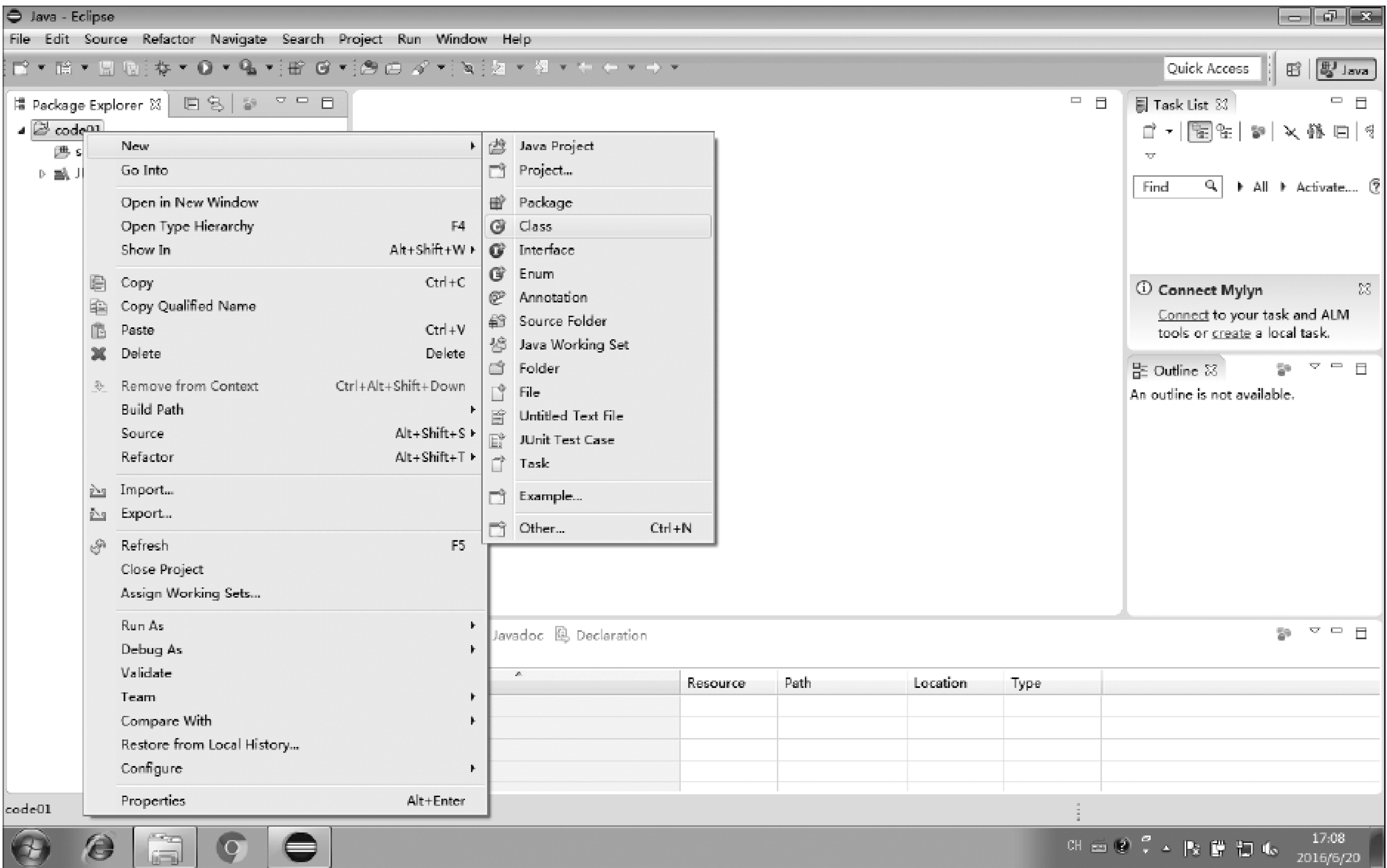


图 1-11　新建 Java 类

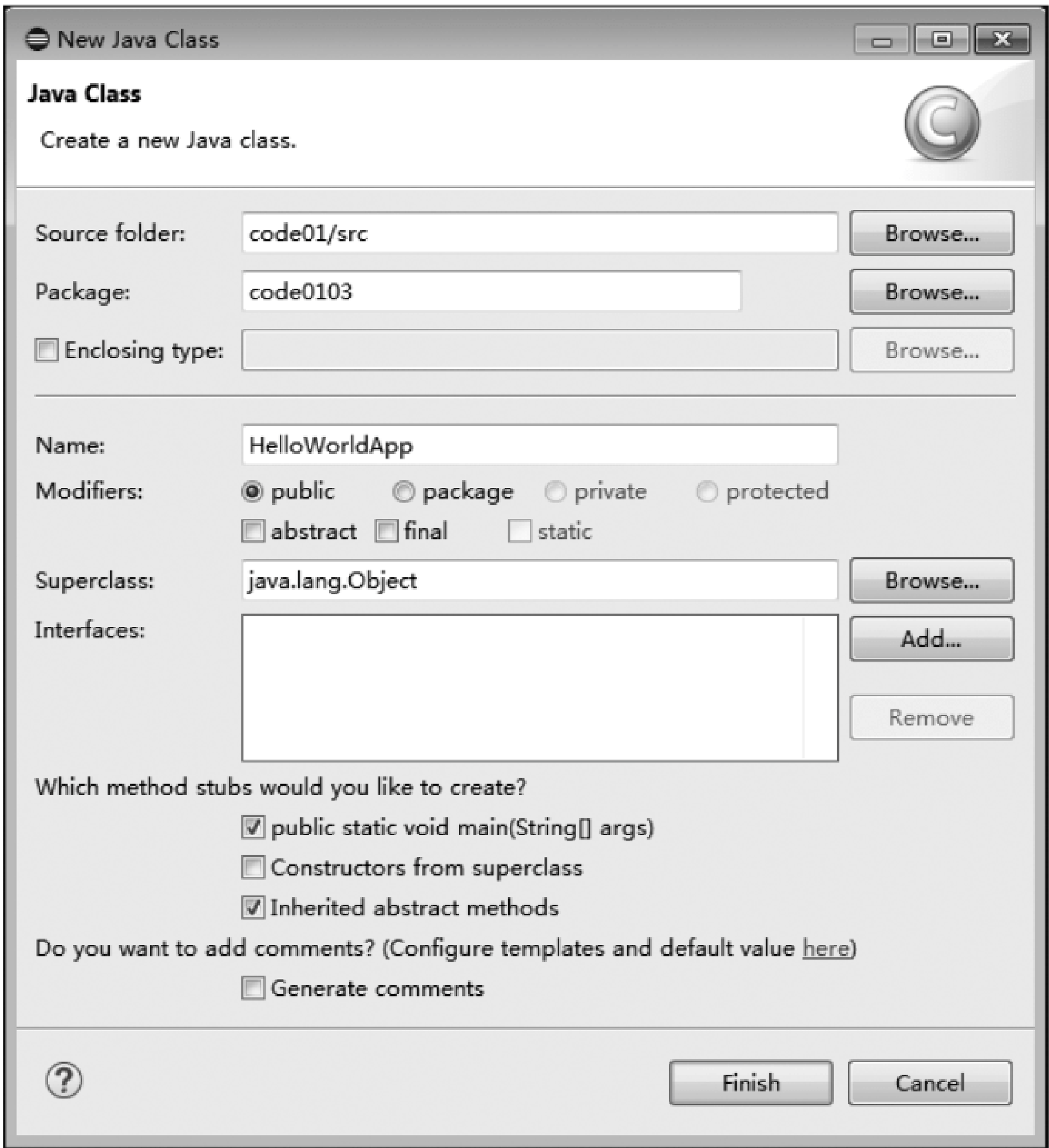


图 1-12　新建 Java 类对话框

(4) 运行 Java 程序。

在新建的 HelloWorld 类中编辑如图 1-13 所示的代码。选中 HelloWorldApp 类,选择工具栏上的图标(如图 1-14 所示),选择 Run as → Java Application,即可运行 HelloWorldApp 程序。程序运行的结果会显示在界面的下方,如图 1-15 所示。

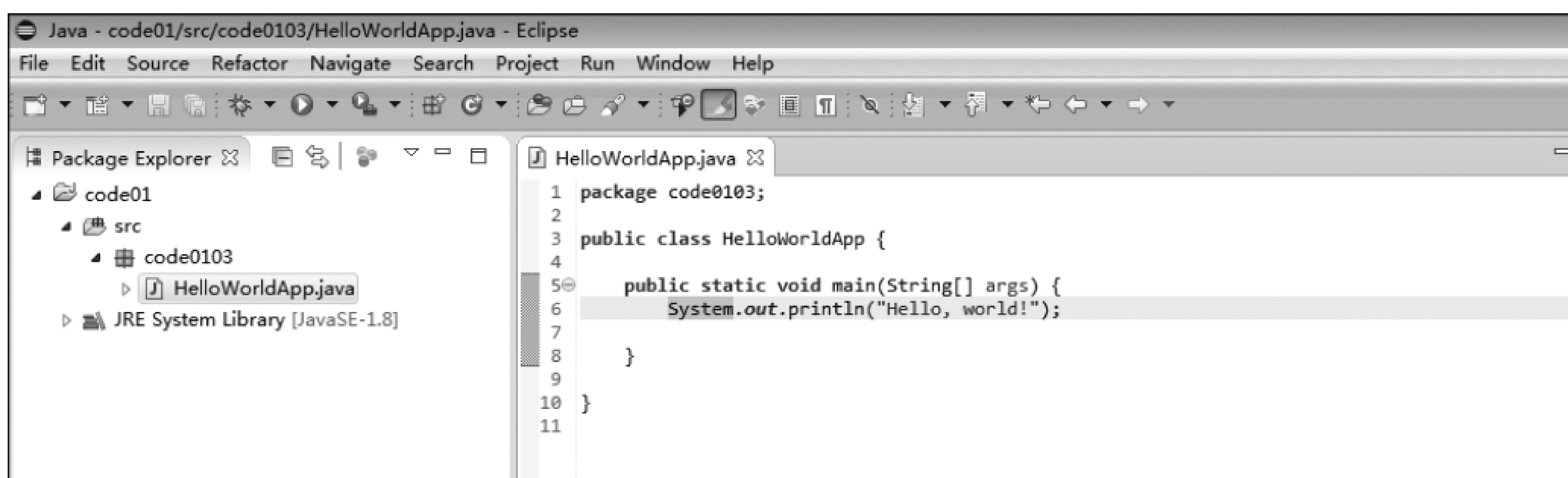


图 1-13 编辑 HelloWorldApp

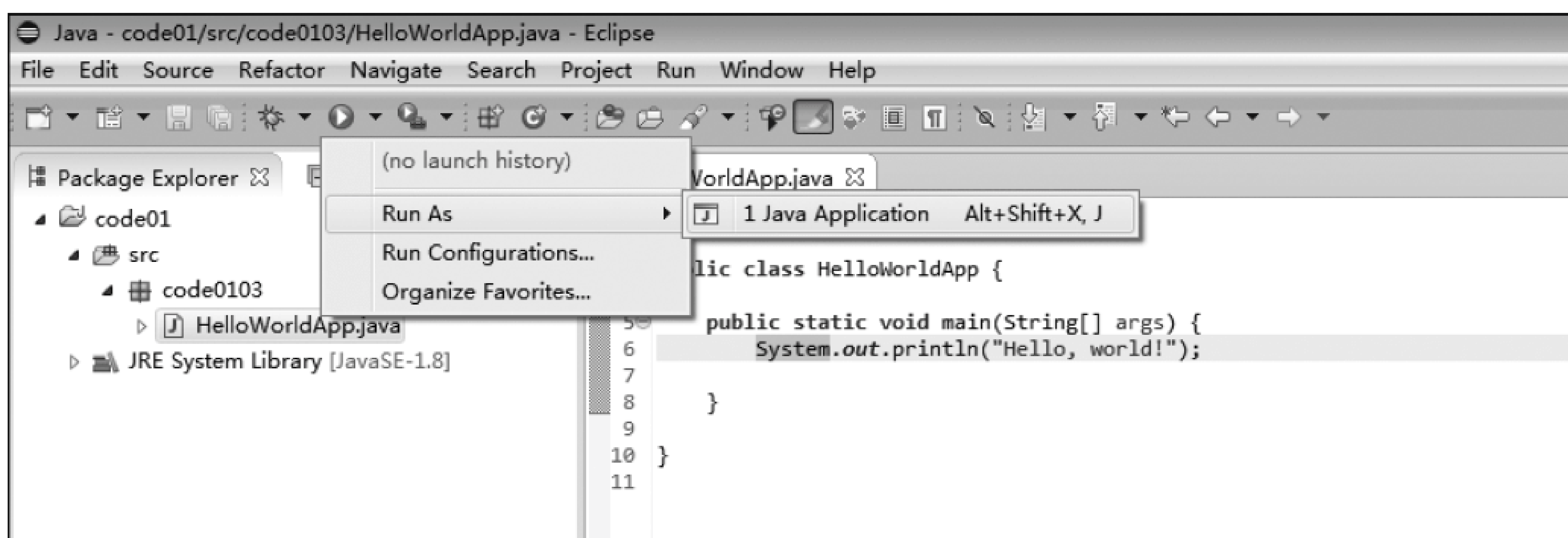


图 1-14 运行 Java 程序



图 1-15 HelloWorldApp 的运行结果

分析:例 1.1 中的程序代码展示了 Java 应用程序的主要结构,主要由 3 部分组成:HelloWorldApp 类的定义、main()方法和注释。

① HelloWorldApp 类的定义:用 class 关键字来声明一个新的类,其类名为 HelloWorldApp,它是一个公共类(用 public 修饰)。整个类的定义由大括号{}括起来。

② main()方法:该方法的声明形式是:

```
public static void main(String[] args)
```

其中,访问权限 public 指明所有的类都可以使用这一方法;static 指明该方法是一个类方法,它可以通过类名直接调用;void 则指明 main()方法不返回任何值;String[] args 是 main()方法接收的参数,其类型是字符串数组,它使得运行时系统可以向 Java 应用程序传

递参数。Java 应用程序以 main()方法作为入口来执行程序,因此对于 Java 应用程序来说,main()方法是必需的。在 Java 程序中,可以定义多个类,每个类中可以定义多个方法,但是最多只有一个公共类,main()方法也只能有一个。在 main()方法的实现(大括号内)中,只有一条语句:

```
System.out.println("Hello, world!");
```

它的作用是将字符串输出到标准输出(命令行窗口)中。

③ 注释://后的内容为注释。注释在程序编译时将被忽略掉,但它可以帮助程序员更好地理解程序代码。

提示:Java 源程序是区分大小写的。公共类必须放在与其同名的文件中。

1.3.2 用 Eclipse 编写带命令行参数的应用程序

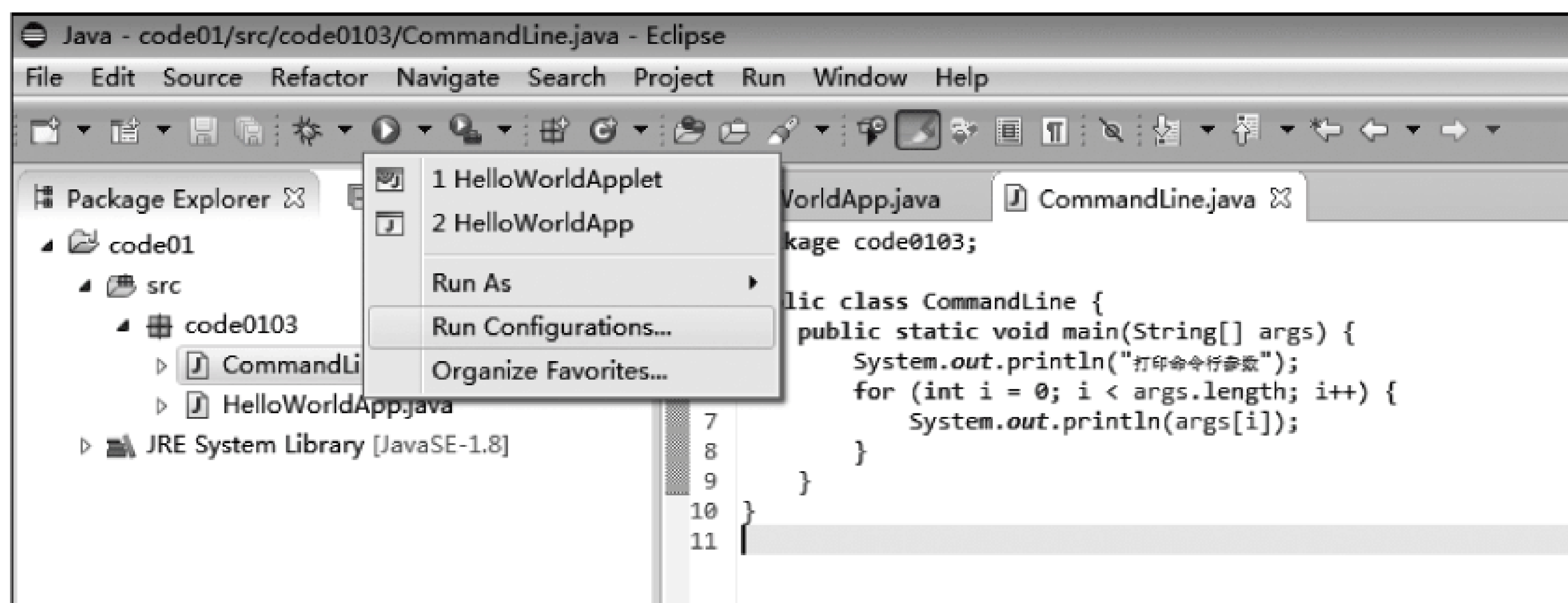
【例 1.2】 用 Eclipse 编写一个带命令行参数的应用程序。

首先在 Eclipse 中编写如下所示的应用程序。

```
package code0103;

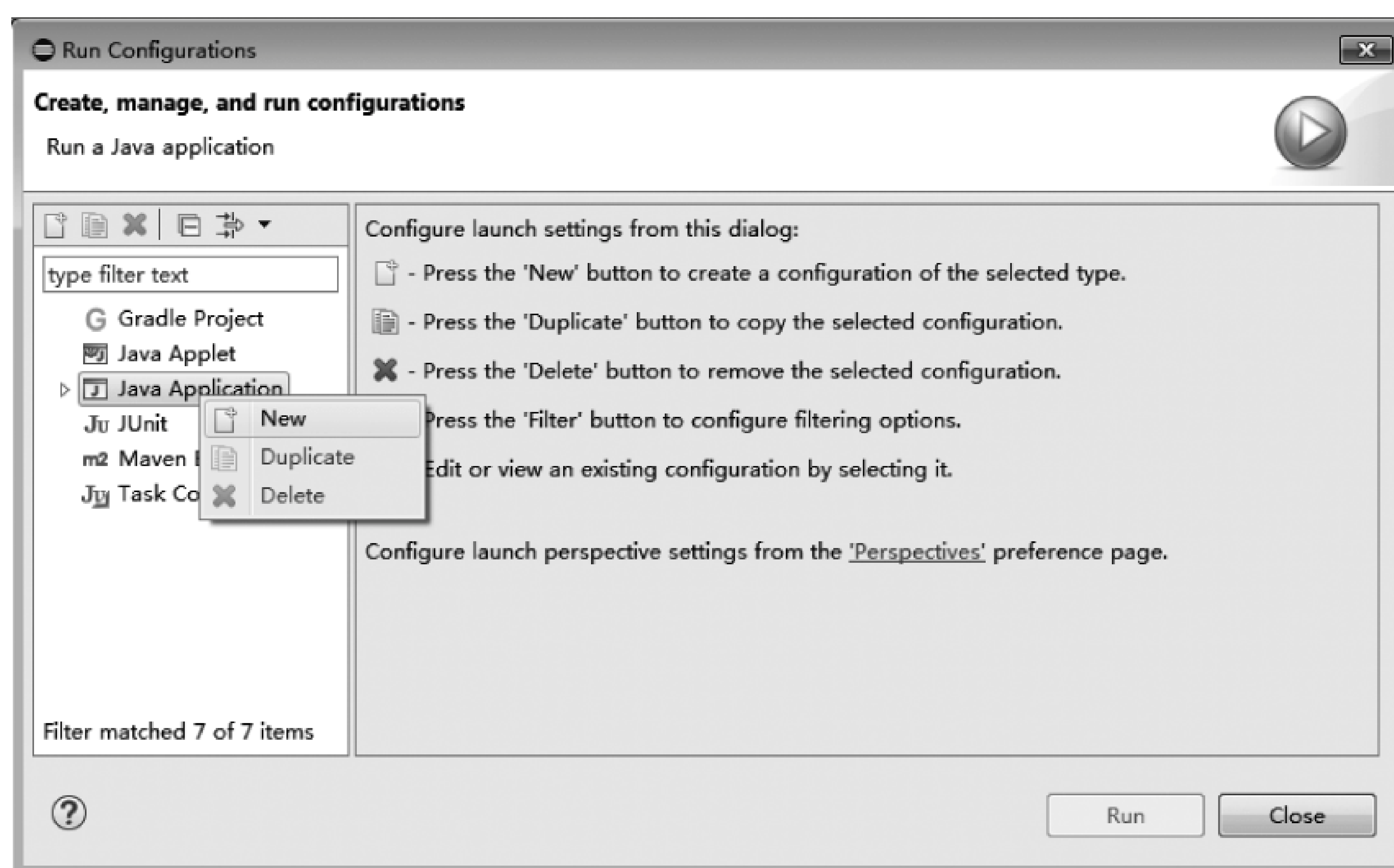
public class CommandLine {
    public static void main(String[] args) {
        System.out.println("打印命令行参数");
        for (int i=0; i<args.length; i++) {
            System.out.println(args[i]);
        }
    }
}
```

运行上述程序的方法与普通应用程序略有区别。首先如图 1-16(a)所示选择 Run Configuration 选项。在图 1-16(b)所示界面中选择 Java Application,在右键菜单中选择 New 选项,新建一个名称为 CommandLine 的 Java 应用程序运行配置。修改 CommandLine 的配置,如图 1-16(c)所示。在 Arguments 选项卡中选中 Program arguments 选项,并在其中输入命令行参数。单击 Run 按钮就可以运行该程序。程序运行的结果如图 1-16(d)所示。

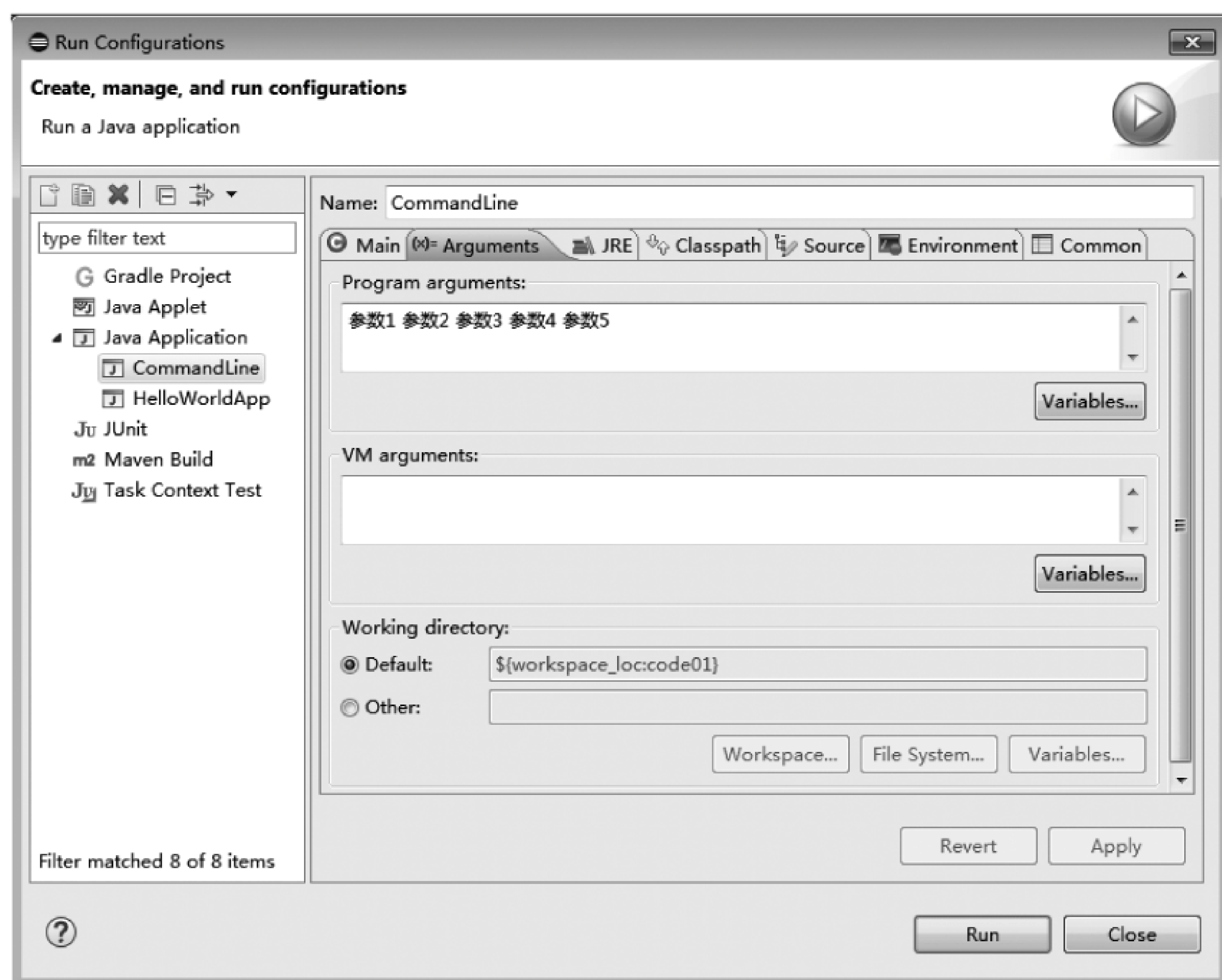


(a)

图 1-16 运行带命令行参数的应用程序


Run Configurations
Create, manage, and run configurations
Run a Java application
type filter text
Gradle Project
Java Applet
Java Application
JUnit
Maven
Task Co
New
Duplicate
Delete
Configure launch settings from this dialog:
- Press the 'New' button to create a configuration of the selected type.
- Press the 'Duplicate' button to copy the selected configuration.
- Press the 'Delete' button to remove the selected configuration.
Press the 'Filter' button to configure filtering options.
Edit or view an existing configuration by selecting it.
Configure launch perspective settings from the 'Perspectives' preference page.
Filter matched 7 of 7 items
Run
Close

(b)


Run Configurations
Create, manage, and run configurations
Run a Java application
type filter text
Gradle Project
Java Applet
Java Application
CommandLine
HelloWorldApp
JUnit
Maven Build
Task Context Test
Filter matched 8 of 8 items
Name: CommandLine
Main
Arguments
JRE
Classpath
Source
Environment
Common
Program arguments:
参数1 参数2 参数3 参数4 参数5
Variables...
VM arguments:
Variables...
Working directory:
Default:
${workspace_loc:code01}
Other:
Workspace...
File System...
Variables...
Revert
Apply
Run
Close

(c)


Problems
Javadoc
Declaration
Console
<terminated> CommandLine [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (2016年6月21日 下午11:39:06)
打印命令行参数
参数1
参数2
参数3
参数4
参数5

(d)

图 1-16 (续)

分析：main 函数的 args 参数就是命令行参数，它使得在运行程序时可以传递数据。命令行参数的类型是字符串数组（String[]）。通过 Eclipse 运行时，需要配置运行参数，添加待传入的命令行参数值。

习题与思考

1. Java 语言的特点是什么？
2. 开发和运行 Java 应用程序需要经过哪些主要步骤？
3. 如何安装和配置 JDK？
4. 应用程序与小应用程序的主要区别是什么？

第 2 章 Java 程序设计基础

本章学习目标

一个 Java 程序是由一个或多个类组成的，类是由最基本的标识符、变量、运算符和语句等组成。本章将介绍 Java 编程中这些最基本元素的定义和使用，以及它们所构成的简单程序。通过本章学习，重点掌握以下内容：

(1) 变量的定义和使用。

(2) 基本数据类型和运算符的使用。

(3) 数据类型的转换。

(4) 程序流程控制语句的使用。

2.1 标识符和关键字

2.1.1 如何定义标识符

就像现实世界中的任何事物都可以有自己的名字一样，在程序设计中，应该为程序中的各个元素进行命名，这种命名的记号就是标识符(Identifier)。在 Java 中，如何正确地定义一个标识符呢?

一般地，在 Java 中标识符是以字母、下划线(_)或美元符号($)等开始的一个字符序列，后面可以跟字母、下划线、美元符号或数字等字符，不能包含运算符和一些特殊字符，如#、*等。

Java 语言使用 Unicode 字符集，一般用 16 位二进制表示一个字符，并且在 0～255 编码区与 ASCII 字符集是兼容的。为了使所有的字符都可以用 16 位二进制表示，Unicode 定义了一种称为 UTF-16 的编码格式，Java 语言中的 char(字符)类型就是用它来表示字符的。

在 Java 中，标识符是大小写敏感的，没有最大长度的限制，不能和关键字相同。下面是一些合法的标识符：

```
Body, _test, $hello
```

一些非法标识符：

```
5Test, hello* , world#, class
```

错误的原因：第一个标识符以数字开始；第二个标识符包含运算符 *；第三个包含特殊符号 #，第四个是关键字。

提示： 在中文环境下，可以使用汉字作为标识符。例如，int 汉字=5；其中，"汉字"就是一个合法的标识符。

2.1.2 关键字

关键字是 Java 中具有特殊含义的字符序列，是程序的重要组成部分。Java 不允许对关键字赋予别的含义。所有的关键字都是小写的，如果被大写就不是关键字了。

1. 定义数据类型的关键字

byte、short、int、long、float、double、char、boolean。

2. 流程控制的关键字

if…else、switch、case、default、do…while、for、break、continue。

3. 方法、类型、变量的修饰关键字

private、public、protected、final、static、abstract、synchronized、volatile。

4. 异常处理关键字

try、catch、finally、throw、throws。

5. 对象相关关键字

new、extends、implements、class、instanceof、this、super。

6. 字面常量关键字

false、true、null。

7. 方法相关关键字

return、void。

8. 包相关关键字

package、import。

提示： 与 C 不同，Java 中无 sizeof 关键字，所有基本数据类型的长度是确定的，不依赖执行环境，所以不需要此关键字。

2.1.3 注释

Java 允许在源程序文件中添加注释，以增加程序的可读性，系统不会对注释的内容进行编译。有以下 3 种形式的注释。

1. 单行注释

单行注释以//开头，至该行结尾，其格式如下：

```
//注释内容
```

2. 多行注释

多行注释以/*开始，遇到*/结束，其格式如下：

```
/* 注释文本
    ⋮
 */
```

3. 文档注释

文件注释以/**开头,遇到*/结束,在注释中每行以*开始,其中可以使用以@开始的特殊标记(tag),注明其后面的文本的含义。例如,@author 后面的文本是"作者"。JDK API 文档就是用此种方式从其源代码产生的。其格式如下:

```
/**注释文本
* @author 张三
* …
*/
```

2.2 常量和变量

2.2.1 常量

Java 中的常量定义和变量的定义基本上是相同的,在类型的前面加上修饰符 final 关键字就可以了。例如:

```
final int MAX=100;
```

定义了一个 int 型的常量 MAX,第一次赋初值后,在程序中就不能再修改它了。一般地,Java 约定常量标识符全部用大写字母,如果由多个单词组成,每个单词大写,用下划线连接。例如:

```
final  int  MAX_COUNT=15;
```

思考:请读者把 Java 定义常量的方式和 C/C++ 语言作比较。

2.2.2 变量

变量是 Java 程序中的基本存储单元,它的值可以修改。变量的定义包括类型、变量名和值等 3 个部分。其中,变量名必须是一个合法的标识符,变量的类型决定了变量的数据性质、范围和存储在内存中所占的字节数以及可以进行的合法操作。其定义格式如下:

```
[修饰符]<类型名>  <变量名>[=<初值>][,<变量名>[=<初值>]… .];
```

其中,<>表示一个占位符,需要替换成具体的类型名称和变量标识符。变量名的长度没有限制。[]中的内容是可选项。例如:

```
int i;
public int j=5, k=4;
```

第一个语句声明了 int 类型的变量 i,声明后,系统将给变量分配内存空间。

第二个语句,在同一行中连续声明了两个 public int 型的变量 j 和 k,并且分别为它们赋了初值,每个变量之间用逗号分隔。

变量的作用域指明该变量能够被访问到的有效范围。声明一个变量的同时也就指明了变量的作用域。按作用域分,变量可分为局部变量、类成员变量和方法参数。

局部变量是在方法内部或代码块中声明的变量，它的作用域为它所在的代码块，在程序设计中，一般以{}为界。

类成员变量的作用域是整个类(参见 4.3.1 节)；方法参数的作用域是它所在方法的方法体(参见 4.3.2 节)。

在一个作用域中，变量名应该是唯一的。在一个作用域中，如果有多个同名的变量可以访问，则按照“邻近”原则，在当前域中定义的变量隐藏其他同名的变量。

【例 2.1】 变量的使用。

```
package code0202;
public class HelloWorldApp {
    int j=5;                                    //此处 j 为类成员变量
    public static void main(String[] args) {
        double j=10;                            //此处 j 为方法 main 的局部变量
        System.out.println("Hello, world!");
        System.out.println(j);                  //输出 double 类型的局部变量 j
    }                                           //double j 作用域结束
}                                               //int j 作用域结束
```

程序运行结果：

```
Hello, world!
10
```

分析：第一个变量 j 定义在所有方法(包括 main)之外，因此它是一个类的成员变量；在 main 方法中定义的第二个同名变量 j，其作用域是方法体的{ }之间，根据“邻近”原则，它将隐藏其他同名变量，因此输出的 j 值应该是局部变量 j 的值。

2.3 基本数据类型

Java 是一种强类型的语言，要求每个变量、每个表达式都必须有确定的数据类型。Java 编译器对所有的表达式和参数都要进行类型相容性的检查，以保证类型是兼容的。任何类型的不匹配都是错误的，在编译器完成编译以前，错误必须改正。

Java 的数据类型可以分为两类：基本数据类型(primitive type)和引用类型(reference type)。详细的分类如图 2-1 所示。

Java 的基本数据类型都有固定的长度，不随运行平台的变化而变化，引用类型都是用类或对象实现的。

提示：Java 不支持 C/C++ 中指针类型、结构类型(struct)和联合类型(union)。

2.3.1 布尔类型

布尔型数据类型用关键字 boolean 表示，只有 true 和 false 两个值，且它们不对应于任何整数值，经常在流程控制语句中使用。

例如：

```
boolean b=false;
```

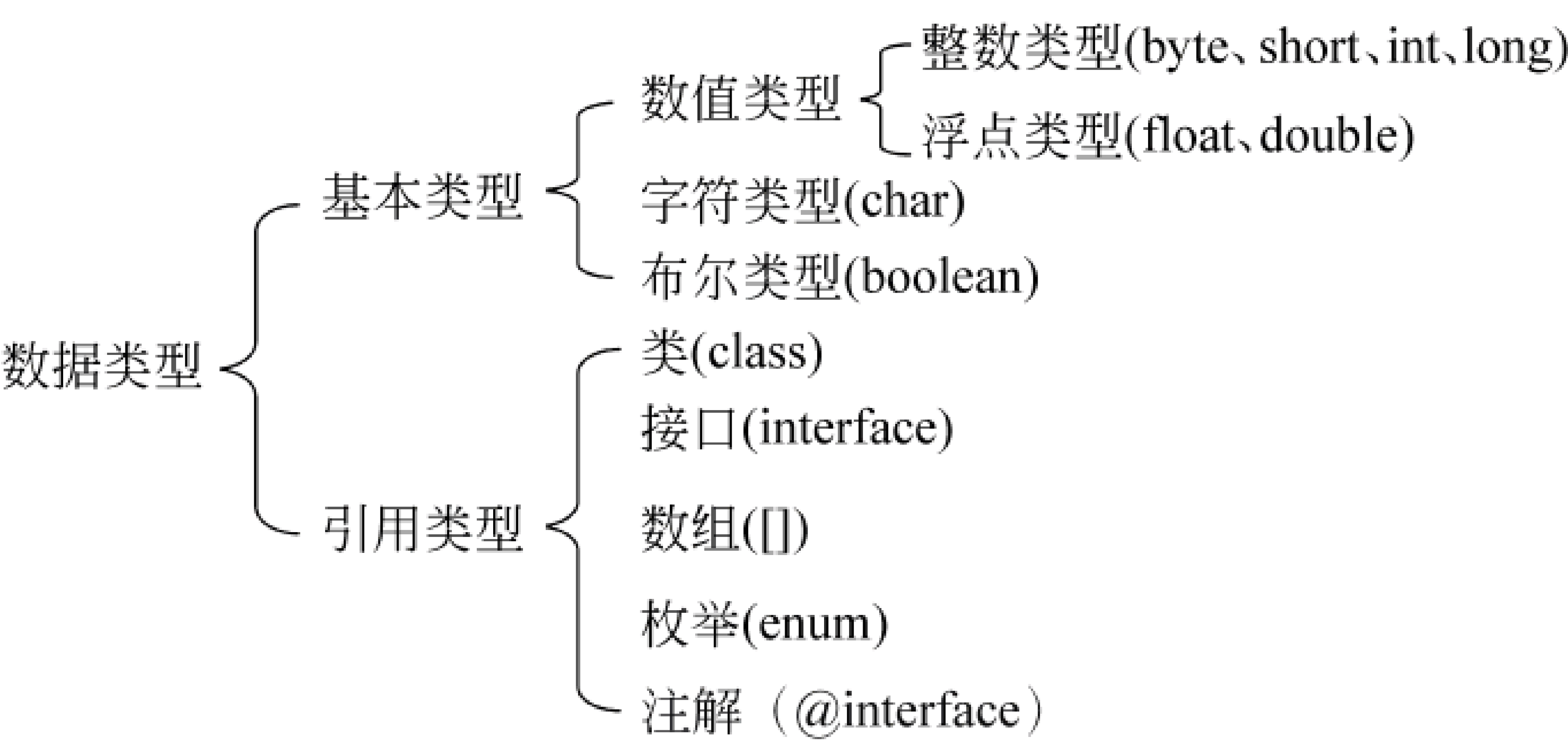


图 2-1　Java 语言的数据类型

2.3.2　字符类型

1. 字符常量

字符常量是用单引号括起来的一个字符，例如：'A'。用双引号括起来的是字符串，例如："hello world"。一个字符也可以用一个 16 位的 Unicode 码表示。

Java 提供了转义字符，以反斜杠(\)开头，如表 2-1 所示。

表 2-1　Java 中的转义字符

转 义 字 符	描　　述
\ddd	1～3 位八进制数所表示的字符(ddd)
\uxxxx	1～4 位十六进制数所表示的字符(xxxx)
\'	单引号字符
\"	双引号字符
\\	反斜杠
\r	回车
\n	换行
\f	走纸换页
\t	横向跳格
\b	退格

例如：

'A'的 Unicode 码是\u0061，那么在 Java 程序中'\u0061'就表示字符'A'。

'中'的 Unicode 码是\u4e2d，那么在 Java 程序中'\u4e2d'就表示字符'中'。

2. 字符变量

字符类型变量用 char 表示，在 Java 虚拟机中一般用 16 位表示一个 char 值，范围为 0～65 535。字符型变量定义格式如下：

```
char  c,  c1='a';
```

其中定义了两个字符型变量 c 和 c1，且 c1 的初始值为'a'。

Java 的字符型数据不同于整数，但是可以和整数在一起运算，从字符型向整数型发生自动类型转换，从整数向字符型转换时需要进行强制类型转换。例如：

```
int i=20000;
char one='1';
int j='a';              //由字符向整数,自动类型转换
char c=(char)(i+one+j);
```

在第 4 条语句中，字符型和整数在一起运算，字符型变量首先转换为整型，这样 i＋one＋j 的运算结果为 20 146，再通过强制类型转换成字符型，字符变量 c 的值为中文字符'亲'。

2.3.3 整数类型

Java 中的整数有 3 种进制形式表示。

（1）十进制：用 0～9 的数表示，首位不能为 0，如 124，－100。

（2）八进制：以 0 开头，后跟多个 0～7 之间的数字，如 0134。

（3）十六进制：以 0x 或者 0X 开头，后跟多个 0～9 之间的数字或 A～F 之间字母的大小写形式。a～f 或者 A～F 分别表示 10～15，如 0x23FE 等于十进制数 9214。

（4）二进制：以 0b 或者 0B 开头，后跟多个 0～1 之间的数字。

Java 定义了 4 种整数类型，如表 2-2 所示。

表 2-2 整数类型

数据类型	所占位数	数的范围
byte	8	$-2^7 \sim (2^7-1)$
short	16	$-2^{15} \sim (2^{15}-1)$
int	32	$-2^{31} \sim (2^{31}-1)$
long	64	$-2^{63} \sim (2^{63}-1)$

一个整数数字隐含为 int 型，在表示 long 型常量时，需要在数字后面加上后缀 L 或 l。例如，3L 表示一个 long 型的常量，而不是 int 型常量。

整数变量定义如下：

```
byte        b=5;
int         i=300;
long        j=300;          //把一个 int 型值赋给 long 型变量 j
long        j2=300L;        //把一个 long 型常量赋值给 long 型变量 j2
int         i=4L;           //错误,不能把 long 型值赋给 int 型变量
```

提示：在定义变量时，所赋值不能超过数据类型的表示范围。如变量 b 的值不能超过 127。

2.3.4 浮点类型

Java 用浮点数表示数学中的实数，即有整数部分和小数部分。浮点数有以下两种表现形式。

(1) 标准计数法：由整数部分、小数点和小数部分组成，如 2.0，345.789。

(2) 科学计数法：由十进制数、小数点、小数和指数构成，指数部分由字母 E 或 e 跟上正负号的整数表示，如 345.789 可以表示成 3.45789E+2。

Java 有两种浮点数类型：单精度浮点数 float 和双精度浮点数 double，如表 2-3 所示。

表 2-3 浮点数类型

数 据 类 型	所 占 位 数	数 的 范 围
float(单精度浮点数)	32	3.4e−038～3.4e+038
double(双精度浮点数)	64	1.7e−308～1.7e+308

一个浮点数字数默认为 double 型。在一个浮点数后加字母 F 或 f，表示 float 型。常量值 3.45 的类型是 double；3.45F 的类型是 float。

浮点数的定义如下：

```
double   d;          //定义一个 double 型变量 d
double   d1=3.4;     //可以在定义的时候赋予初始值
double   d2=3.4d;    //在定义 double 变量时,可以加后缀 D 或 d,也可以不加
float    f=3.4F;     //在定义 float 型变量时,需要在数值后面加 F 或 f
float    f1=3.4;     //默认情况下,常量值 3.4 为 double,编译时会发生类型不匹配的错误
```

2.3.5 各类型数据间的相互转换

各种数据类型的数据可以在一起进行混合运算。运算时，不同类型的数据先转换为相同类型的数据再进行运算。数据类型之间的转换分为自动类型转换和强制类型转换两种。

1. 自动类型转换

从低级到高级发生自动类型转换，也就是从表示范围小的类型向表示范围大的类型发生自动类型转换。不同数据类型的转换如下所示：

低————————————————→高

byte、short、char→int→long→float→double

提示：byte、short 和 char 在一起运算时，首先转换为 int 类型进行运算。

【例 2.2】 分析下面程序中的错误。

```
byte      b1=5;
short     s1=6;
short     s2;
s2=b1+s1;
```

分析：byte 类型和 short 类型的数据进行运算时首先都转换为 int 类型。b1+s1 的结果类型为 int，在第 4 行中就会发生赋值类型不匹配的编译错误，int 型的值不能赋给 short 型的变量 s2。

2. 强制类型转换

由高级向低级数据类型转换时需要强制类型转换，在变量前面，把需要转换的“目标类

型”放到用圆括号()里面。例如：

```
int     i=65;
char    c;
c=(char)i;        //把 int 型变量转换成 char 型,需要强制类型转换
```

2.4 运 算 符

运算符负责对数据进行计算和处理。按运算符所操作数据的个数,可将其分为一元运算符,二元运算符和三元运算符,如++(自增运算)运算符为一元运算符,* 运算符为二元运算符,条件运算符为三元运算符。如果按操作数的类型分,又可将其分为算术运算符、赋值运算符、自增/自减运算符、条件运算符、位运算符、关系运算符和逻辑运算符等。通过运算符可以将各操作数连接成一个有意义的式子,这就是表达式,如 a+6,b*7-9 等。

运算符一般由一个或多个符号构成,如+、>=、<<=。少数运算符有两种含义,应根据上下文理解,如-3 中的-是作为一元运算符(负号)使用,而 a-6 中的-是作为二元运算符(减号)使用。

运算符有优先级。如()的优先级最高,而=的优先级最低,当一个表达式中有多个运算符时,先计算优先级较高的,再计算优先级较低的。运算符也有结合性。

2.4.1 算术运算符

算术运算符主要用于整型或浮点型数据的运算。算术运算符如表 2-4 所示。

表 2-4 算术运算符

	运算符	用法	含义	结合性
二元运算符	+	op1+op2	加法	左
	-	op1-op2	减法	左
	*	op1 * op2	乘法	左
	/	op1/op2	除法	左
	%	op1%op2	模运算(求余)	左
一元运算符	+	+op1	正数	右
	-	-op1	负数	右
	++	++op1,op1++	自增	右,左
	--	--op1, op1++	自减	右,左

1. 算术运算符的运算特点

(1) 对于二元运算符,运算结果的数据类型一般为两个操作数中表达范围较大的类型。例如,一个整数和浮点数运算的结果为浮点数。

(2) 对于一元运算符,运算结果的类型与操作数的类型相同。

(3) 自增、自减运算符有前缀和后缀两种形式,当是前缀形式(即++、--符号出现在变量的左侧)时,对变量实施的运算规则是“先运算后使用”;当是后缀形式(即++、--符号出现在变量的右侧)时,对变量实施的运算规则是“先使用后运算”。

2. 算术运算符的注意事项

(1) 在 Java 中,%(求模运算符)的操作数可为浮点数,如 52.3%10=2.3。

(2) Java 对+运算进行了扩展,可作字符串连接运算符,如"ab"+"efd"得"abefd"。

(3) 做+运算时,如果一个操作数是字符串,其他操作数自动转换成字符串,例如:

```
String s; s="s:"+6*5;          //结果是 s="s:30"
```

(4) byte、short 和 char 等类型进行混合运算时,会先自动转换为 int 类型再运算。

【例 2.3】 算术运算符举例。

```
package code0204;

public class TestOperator{
    public static void main(String[] args) {
        int   i=29;
        int   j=3;
        float     a=23.5f;
        double   b=4.0;
        System.out.println("i+a="+(i+a));          //整数与浮点数相加
        System.out.println("i*j="+(i*j));          //两个整数相乘
        System.out.println("i/j="+(i/j));          //对于整数,运算结果为整数
        System.out.println("i%j="+(i%j));          //求余数
        System.out.println("a*b="+(a*b));          //两个浮点数相乘
        System.out.println("a/b="+(a/b));          //对于浮点数,运算结果为浮点数
        System.out.println("a%b="+(a%b));          //浮点数求余,结果为浮点数
        System.out.println("i++="+(i++));          //先使用,后自增
        System.out.println("++i="+(++i));          //先自增,后使用
    }
}
```

程序的输出结果如下:

```
i+a=52.5
i*j=87
i/j=9
i%j=2
a*b=94.0
a/b=5.875
a%b=3.5
i++=29
++i=31
```

分析: 在打印输出语句中,+除了可以进行整型或浮点数的加法运算外,还可以进行字符串的连接,可以把字符串和数值类型在一起进行运算,最后转换为字符串。

2.4.2 赋值运算符

赋值运算符是二元运算符，左边的操作数必须是变量，右边的操作数为表达式，左右两边的类型如果一致，则直接将右边的值赋给左边的变量；如果不一致，则将表达式的值需要转换为左边变量的类型，再赋值。与其他运算符相比，赋值运算符的优先级最低，且具有右结合特性。

1. 基本赋值运算符

赋值运算符的作用是使变量获得值，基本使用格式如下：

```
<变量名>=<表达式>
```

其中，＝是赋值运算符，＜变量名＞获得计算出的表达式的值。例如：

```
int i,j;        i=10;        j=i+20;
```

2. 扩展赋值运算符

在赋值运算符＝前面加上其他运算符，即构成扩展赋值运算符，例如：

a＋＝5；等价于 a＝a＋5；。

a＊＝b＋5；等价于 a＝a＊(b＋5)，因为赋值运算符的优先级最低。

在 Java 中，大部分的运算符都可以加到＝的前面，构成扩展赋值运算符，常见的扩展运算符如表 2-5 所示。

表 2-5 常见的扩展赋值运算符

运　算　符	示　　例	含　　义
＋＝	count＋＝3	count＝count＋3
－＝	count－＝3	count＝count－3
＊＝	count＊＝3	count＝count＊3
/＝	count/＝3	count＝count/3
％＝	count％＝3	count＝count％3

3. 赋值相容

如果变量的类型和表达式的类型是相同的就可以赋值，称为类型相同；如果两者类型不同，并且变量类型比表达式类型长时，系统会自动将表达式的结果转换为较长的类型。如 int 转换为 long，这时也可以赋值，称为赋值相容(assignment compatible)。例如：

```
long value=98L;          //类型相同
long value2=4;           //int 向 long 自动转换,赋值相容
```

如果变量类型比表达式类型短，则赋值不兼容，编译时产生“可能存在的精度丢失”的错误。例如：

```
int i=99L;               //不能把 long 数据赋值给 int 型变量
```

赋值不兼容时，可以使用强制类型转换，其格式如下：

```
(<目标类型>)<表达式>
```

例如：

```
int i= (int)123123123123L;          //强制类型转换
```

强制类型转换可能会发生精度丢失。

2.4.3 条件运算符

条件运算符由？和：组成，格式如下：

```
(boolean_expr) ? true_statement: false_statement;
```

其含义为：若 boolean_expr 为真，则执行语句 true_satatement；若 boolean_expr 为假，则执行语句 false_statement，而且两条语句需要返回相同(相容)的数据类型。true_statement 和 false_statement 都需要有返回值。

例如：

```
int result=sum==0 ? 100: 2* num;
```

如果 sum 等于 0，则 result 赋值为 100，否则赋值为 2 * num。

条件运算符可以替代简单的 if-else 语句。

2.4.4 位运算符

Java 提供位操作运算符。所有的数在 Java 虚拟机中都会转换为补码二进制表示。

例如：

整数 1 表示为补码二进制为 00000000 00000000 00000000 00000001(4 个字节)。

整数－1 表示为补码二进制为 11111111 11111111 11111111 11111111(4 个字节)。

位运算不是对整个数进行运算，而是对该数的二进制位上的 0 或 1 进行运算。位运算符如表 2-6 所示。

表 2-6　位运算符

运　算　符	示　　例	含　　义
&	op1 & op2	使 op1 和 op2 按位相与
|	op1|op2	使 op1 和 op2 按位相或
～	～op	对 op 按位取反
^	op1^op2	使 op1 和 op2 按位异或
<<	op1<<op2	使 op1 左移 op2 位，右补 0
>>	op1>>op2	使 op1 右移 op2 位(左边补充符号位)
>>>	op1>>>op2	使 op1 无符号右移 op2 位(左边始终补填 0)

位运算符的运算表如表 2-7 所示。

表 2-7 位运算符的运算表

A	B	A&B	A|B	A^B	~A	~B
0	1	0	1	1	1	0
1	1	1	1	0	0	0
0	0	0	0	0	1	1

注意事项：

(1) 除 ~ 为右结合外，其余为左结合。

(2) 操作数的类型一般为整型或字符型。

(3) 若两个数据的长度不同，如 a&b，a 为 byte 型，b 为 int 型，系统首先会将 a 的左侧 24 位填满。若 a 为正，则填满 0；若 a 为负，填满 1，即进行“符号扩充”。

下面通过示例详解位运算符：

1. 按位与运算符 &

例如：5&9，其结果等于 1。

&	十进制数	二进制数
操作数 1	5	00000000 00000000 00000000 00000101
操作数 2	9	00000000 00000000 00000000 00001001
运算结果	1	00000000 00000000 00000000 00000001

除了末位，其他位都清 0。

2. 按位或运算符 |

例如：5|9，其结果等于 13。

3. 按位异或运算符^

例如：5^9，其结果等于 12。

4. 按位取反运算符~

例如：~5，其结果等于−6。

5. 左移运算符<<

例如：5<<2，其结果等于 20。

6. 右移运算符>>

例如：5>>2，其结果等于 1。

7. 无符号右移运算符>>>

在移动时，高位填 0，最低位被舍弃。

例如：−1>>>1，结果为 2147483647。

思考：(1) 5>>32 的结果是多少？答案是 5。

(2) 5>>34 的结果是多少？答案是 1。

(3) 5L>>64 的结果是多少？答案是 5L。

分析：在移位运算中，对于 int 类型，第二个操作数先对 32 取模，余数是实际移动的位数。对于 long 型，第二个操作数先对 64 取模，余数是实际移动的位数。

2.4.5 关系运算符

关系运算符用来比较两个值之间的大小,结果返回布尔值:true 或者 false。关系运算符有 6 种,如表 2-8 所示。

表 2-8 关系运算符

运 算 符	示 例	含 义
==	op1==op2	比较两个数据是否相等
!=	op1 !=op2	比较两个数据是否不等
<	op1<op2	比较一个是否小于另一个
>	op1>op2	比较一个是否大于另一个
<=	op1<=op2	比较一个是否小于等于另一个
>=	op1>=op2	比较一个是否大于等于另一个

注意事项:

(1) 注意==不要写成=。

(2) Java 中,任何类型的数据(包括简单类型和复合类型),都可以通过==或!=来比较是否相等。

(3) 关系运算的结果是 true 和 false,而不是 1 或 0。

(4) 关系运算符的结果是布尔值,关系运算符的优先级要高于逻辑运算符。

2.4.6 逻辑运算符

逻辑运算又称布尔运算,只能处理布尔类型的数据,所得结果也是布尔值。逻辑运算符主要有 3 种:逻辑与(&&),逻辑或(||)、逻辑非(!)。运算符如表 2-9 所示。

表 2-9 逻辑运算符

x	y	x&&y	x||y	!x	!y
true	true	true	true	false	false
true	false	false	true	false	true
false	false	false	false	true	true

逻辑运算符支持“短路运算”(short-circuit)。所谓短路计算就是从左向右依次计算每个条件是否成立,如果在前面的计算中已经可以得出整个复合条件表达式的计算结果,则后面的条件就不计算了。

例如,int i=5,对于表达式(i>8)&&(i<4)的计算:在这个表达式中,当 i>8 时计算结果为 false,不论后面条件的结果如何,整个复合条件的计算结果都为 false,则后面的条件就无须计算了。

2.5 表　达　式

表达式是程序设计语言的基本组成部分，表示一种求值的规则，是由运算符和操作数组成的符号序列。表达式根据运算符的优先级别和结合性，首先执行指定的计算，再返回某个值。

表达式的运算结果的类型就是表达式的类型。表达式一般用来给变量赋值，以及在程序中作为控制条件。在表达式中使用的变量必须已经被初始化。

如果把一个表达式赋值给某个变量，同样需要进行类型检查，如果两边类型不同，则需要进行强制或自动类型转换。

一个常量或者变量名是最简单的表达式，其值就是该常量或变量的值。

在对表达式进行运算时，遵循一定的规则，要按运算符的优先级从高到低进行，同级的运算符则按从左到右的方向进行。表 2-10 列出了 Java 运算符的优先级。

表 2-10　Java 运算符的优先级

高	1	.　[ ]　()　++　--　!　~　instanceof
↓	2	new (type)
	3	*　/　%
	4	+　-
	5	>>　>>>　<<
	6	<>　<=　>=
	7	==　!=
	8	&
	9	^
	10	|
	11	&&
	12	||
	13	?:
	14	=　+=　-=　*=　/=　%=　^=
低	15	&=　!=　<<=　>>=　>>>=

通过对表 2-10 进行分析，可以得出以下几个结论：

(1) 赋值运算符的优先级最低，因为赋值运算符要使用表达式的值。

(2) 关系运算符的优先级比逻辑运算符的优先级高。

(3) 一元运算符的优先级也比较高。

(4) 算术运算符要比关系运算符和二元逻辑运算符的优先级要高。

在表达式中，为了使表达式的结构更加清晰，可以显式地用()标明运算次序，括号中的

表达式首先被计算。

例如:

```
x<y&&x>10||y>0&x<50|y>50;
```

可以加上括号使表达式的结构更清晰

```
((x<y)&&(x>10))||(((y>0)&(x<50))|(y>50));
```

2.6 程序控制语句

Java 程序控制语句分为 3 类:选择、循环和跳转语句。其中,①选择语句根据条件表达式计算结果,并使程序可以选择不同的执行路径;②循环语句使程序能够重复执行一个或多个语句;③跳转语句允许程序以非线性的方式执行。

2.6.1 if 语句

if 语句是 Java 中的条件分支语句,它能将程序的执行路径分为两条。if 语句的完整格式如下:

```
if(condition)
    statement1;
else
    statement2;
```

其中,if 或 else 后面可以是单个语句(statement),也可以是程序块(block)。条件 condition 是任何返回布尔值的表达式,else 子句是可选的。

if 语句的执行过程如下:如果条件为真(true),则执行 if 后面的语句(statement1);否则,执行 else 后面的语句(statement2)。任何时候两条语句都不可能同时执行。

如果一个 if 语句中包含另外一个 if 语句则为嵌套 if 语句。如下面的例子:

```
if(i==10) {
    if(j<20)
        a=b;
    if(k>100)
        c=d;
    else a=c;          //与上面最近的 if 对应
} else                 //与第一个 if 对应
{
    a=d;
}
```

另外一种嵌套形式是:if-else if 阶梯,它的语法如下:

```
if(condition)
    statement;
else if(condition)
```

```
      statement;
else if(condition)
      statement;
⋮
else
      statement;
```

条件表达式从上到下被求值。一旦找到为真的条件,就执行与它关联的语句,该阶梯的其他部分就被忽略了。如果所有的条件都不为真,则执行最后的 else 语句。

【例 2.4】 通过使用 if-else if 阶梯来确定某个月是什么季节。

```
package code0206;
//IfElse.java
public class IfElseDemo {
    public static void main(String args[]) {
        int month= 9; //Sept
        String season;
        if (month==12 || month==1 || month==2)
            season="Winter";
        else if (month==3 || month==4 || month==5)
            season="Spring";
        else if (month==6 || month==7 || month==8)
            season="Summer";
        else if (month==9 || month==10 || month==11)
            season="Autumn";
        else
            season="It is a wrong number!";
        System.out.println("Sept. is in the "+ season+ ".");
    }

}
```

程序运行结果:

```
Sept. is in the Autumn.
```

2.6.2 switch 语句

switch 语句是 Java 的多路分支语句,它基于表达式的值使程序执行不同语句序列。switch 语句的通用形式如下:

```
switch (expression) {
case value1:
  … //语句序列
  break;
case value2:
  … //语句序列
```

```
        break;
    case valueN:
        … //语句序列
        break;
    default:
        … //默认语句序列
}
```

表达式 expression 的计算结果必须为 byte、short、int、char、字符串或者 enum 类型，每个 case 语句后的值 value 必须是与 expression 类型兼容的一个常量。重复的 case 值是不允许的。

switch 语句的执行过程如下：表达式的值首先与每个 case 语句中的常量进行比较。如果发现了一个与之相匹配的，则执行该 case 语句后的代码，不再匹配后面的 case 语句。如果没有一个 case 常量与表达式的值相匹配，则执行 default 语句。当然，default 语句是可选的。如果没有相匹配的 case 语句，也没有 default 语句，则什么也不执行。

case 语句只是起到一个标号作用，用来查找匹配的入口，并从此处开始执行其后的语句序列。

switch 语句的执行过程和 default 语句的位置没有关系，不会因为把 default 语句放在 switch 的开始处而执行 default 语句。

在 case 语句序列中的 break 语句将使程序执行流从整个 switch 语句退出，程序将从整个 switch 语句后的第一行代码开始继续执行。如果没有遇到 break 语句，switch 语句将一直执行到其结束。

【例 2.5】 switch 语句使用，可以与例 2.4 比较。

```
package code0206;

//SwitchBreak.java
public class SwitchBreakDemo {
    public static void main(String[] args) {
        int month= 9;
        String season;
        switch (month) {
        case 12:
        case 1:
        case 2:
            season= "Winter";
            break;
        case 3:
        case 4:
        case 5:
            season= "Spring";
            break;
        case 6:
        case 7:
```

```
        case 8:
            season="Summer";
            break;
        case 9:
        case 10:
        case 11:
            season="Autumn";
            break;
        default:
            season="Wrong Month";
        }
        System.out.println("Sept is in the "+season+".");
    }
}
```

程序运行结果：

```
Sept is in the Autumn.
```

分析：正如该程序所演示的那样，如果没有 break 语句，程序将继续执行下面的每一个 case 语句，直到遇到 break 语句（或 switch 语句的末尾）。

switch 语句也可以嵌套，即将一个 switch 语句作为一个外部 switch 语句序列的一部分。

2.6.3 while 与 do-while 语句

while 语句是 Java 最基本的循环语句，当它的条件表达式是 true 时，while 语句重复执行循环体，循环体可以是一个语句或者语句块。它的通用格式如下：

```
while(condition) {
    //循环体
    ⋮
}
```

【例 2.6】 使用 while 计算大于 100 且小于等于 200 的数的和。

```
package code0206;
//SampleWhile.java
public class SampleWhile
{
    public static void main(String[] args)
    {
        int sum=0, i=100;
        while (i<200) {
            i++;
            sum+=i;
        }
```

```
            System.out.println("the sum is "+ sum);
        }
    }
```


程序运行结果:

```
the sum is 15050
```

while 循环(或 Java 的其他任何循环)的循环体可以为空。因为一个空语句(仅由一个分号组成的语句)在 Java 的语法上是合法的。例如,下面的程序:

```
int i=100, j=200;
while(++i<--j)
    ; //没有循环体
```

如果 while 循环一开始条件表达式就是 false,那么循环体就根本不被执行。然而,有时需要先执行循环体,再判断条件表达式。Java 提供了 do-while 循环:它的循环体至少执行一次,因为它的条件表达式在循环的结尾。通用格式如下:

```
do {
//循环体
    ⋮
} while(condition);
```

do-while 循环总是先执行循环体,然后再计算条件表达式。如果表达式为 true,则循环继续;否则,循环结束。

【例 2.7】 使用 do-while 计算大于 100 且小于等于 200 的数的和(重写例 2.6)。

```
package code0206;
//SampleDowhile.java
public class SampleDowhile
{
    public static void main(String[] args)
    {
        int sum=0, i=100;
        do {
            i++;
            sum+=i;
        } while (i<200);
        System.out.println("the sum is "+ sum);
    }
}
```

程序运行结果:

```
the sum is 15050
```

2.6.4 for 语句

for 循环是一个功能强大且形式灵活的结构。下面是 for 循环的通用格式:

```
for(initialization; condition; iteration) {
    //循环体
    ⋮
}
```

如果只有一条语句需要重复,大括号就没有必要。for 循环的执行过程如下:

第一步,当循环启动时,先执行其初始化部分。初始化表达式仅被执行一次。通常,这是设置循环控制变量值的一个表达式,作为控制循环的计数器。

第二步,计算条件 condition 的值。

第三步,如果条件表达式为 true,则执行循环体;如果条件表达式为 false,则循环终止。

第四步,执行循环体的迭代(iteration)部分,这部分通常是增加或减少循环控制变量的一个表达式。该步结束后,转到第二步执行。

控制 for 循环的变量经常只是用于该循环,而不用在程序的其他地方。在这种情况下,可以在循环的初始化部分中声明变量。

【例 2.8】 使用 for 计算大于 100 且小于等于 200 的数的和(重写例 2.7)。

```
package code0206;
//SampleFor.java
public class SampleFor
{
    public static void main(String[] args)
    {
        int sum=0;
        for (int i=101; i<=200; i++) {
            sum+=i;
        }
        System.out.println("the sum is "+sum);
    }
}
```

程序运行结果:

```
the sum is 15050
```

Java 允许两个或两个以上的变量控制 for 循环,可以在初始化部分和迭代部分声明和使用多个变量,每个变量之间用逗号分开。请看下面的程序:

```
int a, b;
for (a=1, b=4; a<b; a++, b--) {
    System.out.println("a="+a);
    System.out.println("b="+b);
}
```

for 循环的初始化和迭代部分可以为空。如果 for 循环的 3 个部分全为空,就是一个无限循环(从来不停止的循环)。例如:

```
for(;;) {
```

```
        //循环体
        ⋮
    }
```

这个循环将始终运行。尽管有一些程序,例如操作系统命令处理器,需要无限循环,但大多数“无限循环”实际上是具有特殊终止条件的循环。在后面的 2.6.5 节中将介绍如何使用 break 语句来终止这种类型的循环。

为了更方便地遍历一个数组或集合中的所有元素,Java 提供了 for…each 语句。下面通过例子来说明。

【例 2.9】 使用 for…each,求大于 100 且小于 200 的整数的和。

```
package code0206;

//ForEachDemo.java
public class ForEachDemo {
    public static void main(String[] args) {
        int sum=0;
        int a[]=new int[100];    //初始化数组,参见第 3 章
        for (int i=0; i<100; i++)
            a[i]=101+i;
        //for…each 语句的使用
        for (int e: a)
            sum=sum+e;
        System.out.println("the sum is "+sum);
    }
}
```

该程序中的冒号“:”表示 in 的意思,for(int e:a)就是 for each int e in a,即“对于数组 a 中的每个整数 e”。通过定义一个整数变量 e 表示数组中的每个元素。for…each 循环看上去比一般的 for 循环漂亮得多,不需要使用下标或者循环变量。

2.6.5 break 语句

Java 中,break 语句有 3 种作用:第一,在 switch 语句中,它被用来终止一个语句块;第二,它能被用来退出一个循环;第三,break 后面加语句标签实现跳转。下面对后两种用法进行解释。

可以使用 break 语句直接强行退出循环。在循环中遇到 break 语句时,循环被终止,执行循环后面的语句。下面是一个简单的例子:

【例 2.10】 break 退出循环。

```
package code0206;

//BreakLoop.java
class BreakLoop {
    public static void main(String args[]) {
      for(int i=0; i<100; i++) {
        if(i==5) break;       //如果 i 等于 5,终止循环
```

```
                System.out.println("i: "+i);
            }
            System.out.println("Loop complete.");
        }
}
```

程序运行结果：

```
i: 0
i: 1
i: 2
i: 3
i: 4
Loop complete.
```

分析：正如所看到的那样，尽管 for 循环被设计为从 0 执行到 99，但是当 i 等于 5 时，break 语句终止了程序。

break 语句能用于任何 Java 循环中，包括 while、do…while 循环。在一系列嵌套循环中使用 break 语句时，它仅仅终止最里面的循环。

提示：关于 break 语句，要记住两点。首先，一个循环中可以有一个以上的 break 语句。但要小心，太多的 break 语句会破坏代码结构；其次，switch 语句中的 break 仅仅影响该 switch 语句，而不会影响其外部的任何循环；如果 switch 内嵌有循环语句，则循环语句内的 break 语句跳出循环，而不是终止 switch 语句。

break 语句加标签可以用于从嵌套很深的循环中退出，其定义格式如下：

```
break label;
```

这里，标签 label 是标识代码块的标签。当这种形式的 break 执行时，控制被传递到指定的代码块。被加标签的代码块必须包围此 break 语句，但是它不需要直接包围 break。这意味着可以使用一个加标签的 break 语句退出一系列的嵌套块。但是不能使用 break 语句将控制传递到不包含 break 语句的代码块。

要指定一个代码块，在其开头加一个标签即可。标签(label)可以是任何合法有效的 Java 标识符后跟一个冒号。一旦给一个块加上标签就可以使用这个标签作为 break 语句的对象了。

例如，下面的程序示例了 3 个嵌套循环，每一个都有它自己的标签。break 语句跳出了标签 second 的代码块。

【例 2.11】 带标签的 break 的使用。

```
package code0206;
//BreakLabel.java
class BreakLabel
{
    public static void main(String args[])
    {
        boolean t=true;
```

```
            first: {
                second: {
                    third: {
                        System.out.println("Before the break.");
                        if (t)
                            break second;
                        System.out.println("This won't execute");
                    }
                    System.out.println("This won't execute");
                }
                System.out.println("This is after second block.");
            }
        }
}
```

程序运行结果:

```
Before the break.
This is after second block.
```

2.6.6 continue 语句

有时需要强迫一次循环提前结束从而进行下一次循环。也就是说,可以继续运行循环,但是要忽略这次循环剩余的循环体的语句,这时要用到 continue 语句。

在 while 和 do…while 循环中,continue 语句直接跳转到循环的条件表达式,然后继续循环过程,continue 后面的语句被忽略。

下例使用 continue 语句,在每行打印 5 个数字。

【例 2.12】 continue 的使用。

```
package code0206;

//SampleContinue.java
class SampleContinue
{
    public static void main(String args[])
    {
        for (int i=1; i<20; i++) {
            System.out.print(i+" ");
            if (i % 5 != 0)
                continue;
            System.out.println("");
        }
    }
}
```

程序运行结果:

```
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19
```

分析：该程序使用%(模)运算符来检验变量 i 是否被 5 整除，如果不是，循环继续执行而不输出一个新行；如果能被 5 整除，那么输出一个新行。

类似 break 语句，continue 也可以指定一个标签来说明继续哪个包围它的循环。下面的例子运用 continue 语句来打印 1～9 的三角形乘法表。

【例 2.13】 带标签的 continue。

```
package code0206;

//ContinueLabel.java
class ContinueLabel
{
    public static void main(String args[])
    {
        outer: for (int i=1; i<10; i++) {
            for(int j=1; j<10; j++) {
                if(j>i) {
                    System.out.println();
                    continue outer;
                }
                System.out.print(" "+ (i * j));
            }
        }
        System.out.println();
    }
}
```

程序运行结果：

```
1
2 4
3 6 9
4 8 12 16
5 10 15 20 25
6 12 18 24 30 36
7 14 21 28 35 42 49
8 16 24 32 40 48 56 64
9 18 27 36 45 54 63 72 81
```

2.6.7 return 语句

return 语句用来明确地从一个方法返回。也就是说，return 语句使程序控制权返回到调用它的方法，因此 return 也属于跳转语句。

在一个方法的任何时间,return 语句可以使程序返回到调用它的方法。下面的例子就说明了这一点,由于本例是 Java 运行时系统调用 main(),因此 return 语句使程序执行返回到 Java 运行时系统。

【例 2.14】 return 语句的使用。

```
package code0206;

//ReturnTest.java
class ReturnTest
{
    public static void main(String args[])
    {
        boolean t=true;
        System.out.println("Before the return.");
        if(t)
            return;
        System.out.println("This won't execute.");
    }
}
```

程序运行结果:

```
Before the return.
```

由结果可知,最后的 println("This won't execute.")语句没有被执行。因为一旦 return 语句被执行,程序控制转移到它的调用者。

如果 return 语句需要返回一个值给调用者,则可以使用下列格式:

```
return 返回值
```

如果 return 语句未出现在子方法中,则子方法的最后一条语句执行后自动返回到主方法。

2.6.8 实用案例

【例 2.15】 计算斐波那契数列。

输出斐波那契(Fibonacci)数列的前 40 项。该数列的前两项都是 1,从第 3 项开始,其后的每一个数据项都是前面的两个数据项之和。

本实用案例程序如下:

```
package code0206;

//Fibonacci.java
public class Fibonacci {
    public static void main(String[] args) {
        long f1,f2,f3,n=50;
        f1=f2=1;
        System.out.print(f1+" "+f2+" ");
        for(int i=3;i<=n;i++)
```

```
            {
                f3=f1+f2;          //第 3项是前两项之和
                f1=f2;             //更新前两项 f1和 f2
                f2=f3;
                System.out.print(f3+" ");
                if(i%10==0)        //输出 10项后换行
                {
                    System.out.println();
                }
            }
        }
}
```

程序运行结果：

```
1 1 2 3 5 8 13 21 34 55
89 144 233 377 610 987 1597 2584 4181 6765
10946 17711 28657 46368 75025 121393 196418 317811 514229 832040
1346269 2178309 3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
```

2.7 实训任务

【任务描述】

假定公鸡 5 元钱 1 只，母鸡 3 元钱 1 只，小鸡 1 元钱 3 只。现在有 100 元钱要求买 100 只鸡，请编程列出所有可能的购鸡方案。

【任务分析】

这是一个典型的约束满足问题，即从可能的解空间(3 种鸡的个数)中找到满足要求(鸡的数量等于 100，价格等于 100)的解。对这类问题，基本的做法就是通过循环语句遍历整个解空间，判断当前找到的解能否满足要求，如果满足则输出，否则继续寻找，直至搜索完毕。

假定 i 为公鸡数(i≤20)，j 为母鸡数(j≤33)，k 为小鸡数(k≤300)，如果条件(5＊i+3＊j+k/3==100)和(i+j+k==100)同时成立，则意味着找到一个满足约束的解，否则应该继续试探下一个解，比如将 i 或 j 加 1，或将 k 加 3(加 1 或加 2 都可能出现无法除尽的情况)。

【任务解决】

完整示例代码如下：

```
public class chick{
  public static void main(String[] args){
      int count=0;
      for(int i=0;i<=20;i++)
      {
          for(int j=0;j<34;j++)
          {
              for(int k=0;k<=300;k=k+3)
```

```
                {
                if((5* i+3* j+k/3==100)&&(i+j+k==100))
                  {
                      System.out.println("母鸡:"+i+"公鸡:"+j+"小鸡:"+k);
                      count++;
                    }
                }
            }
        }
        System.out.println("共有"+count+"种买法");
    }
}
```

程序运行结果:

```
母鸡:0 公鸡:25 小鸡:75
母鸡:4 公鸡:18 小鸡:78
母鸡:8 公鸡:11 小鸡:81
母鸡:12 公鸡:4 小鸡:84
共有 4种买法
```

习题与思考

1. 简述基本数据类型和引用数据类型的基本特点。
2. 使用异或运算符^实现两个整数的交换。
3. 编程序,显示螺旋方阵:

1	2	3	4
12	13	14	5
11	16	15	6
10	9	8	7

4. 下列哪个是合法的标识符?(   )
   A. a=b　B. _Hello　C. 2nd　D. Chong qing
5. 下列哪些是合法的标识符?(   )
   A. new　B. class　C. int　D. const2
6. 如果定义变量 double d1, d2=4.0,则下列说法正确的是(   )。
   A. 变量 d1、d2 均初始化为 4.0　B. 变量 d1 没有初始化,d2 初始化为 4.0
   C. 变量 d1、d2 均未初始化　D. 变量 d2 没有初始化,d1 初始化为 4.0
7. 判断题:所有的变量在使用前都必须进行初始化。(   )
8. 内部数据类型 byte 的取值范围是(   )。
   A. 0～65 535　B. －128～127
   C. －32 768～32 767　D. －256～255

9. 下列哪些是不能通过编译的语句？(　　)

A. int i=32；　　　　B. float f=45.0；

C. double d=45.0；　　　　D. char a='c'；

10. 编写一个程序求 1!+2!+…+10!。

11. 编程验证哥德巴赫猜想，任何大于 6 的偶数可以表示为两素数之和，如 10=3+7。

第 3 章 数　组

本章学习目标

数组是由类型相同的元素顺序组成的一个集合，每个数组都有一个唯一的名称，即数组名。通过数组名和下标可以方便地访问数组中的每个成员。本章主要讲述 Java 中一维和二维数组的创建和使用方法。通过本章的学习，应重点掌握以下主要内容：

（1）一维数组的定义和产生方法。

（2）一维数组的初始化方法。

（3）一维数组的引用方法。

（4）二维数组的创建和引用方法。

3.1　数组使用初探

Java 程序中通常使用变量来存放各种类型的数据，如将 10 个数分别存放在 10 个变量中，但这会非常麻烦，因为需要的变量名实在太多了。这时就可以考虑用一个数组变量来存放它们，并通过一个下标来访问存入数组中的每个成员。

【例 3.1】 求 10 个学生某门课程的平均成绩。

分析： 如果用变量 s1，s2，…，s10 来保存 10 个学生的成绩，显然不合适（变量太多且随学生人数递增），而如果使用数组 s 来保存这些成绩，则问题可以处理如下：

```
package code0301;

public class AvgGrade {
    public static void main(String args[]) {
        int total=0;
        int s[]={ 75, 69, 80, 85, 93, 97, 79, 77, 68, 90 };      //定义数组变量 s并赋值
        double avg=0;
        for (int i=0; i<10; i++) {
            total=total+s[i];                                 //访问数组成员 s[i]
        }
        avg=total/10.0;                                       //求平均分 avg
```

```
        System.out.println("The average score is:"+avg);
    }
}
```

程序运行结果：

```
The average score is: 81.3
```

该问题的求解涉及两个关键点：

(1) 定义数组变量 s[]，将所有值{75，69，80，85，93，97，79，77，68，90}存入其中，具体内容参见 3.2.1 节。

(2) 通过数组变量 s 加下标 i 的形式访问数组中每个成员：total＝total＋s[i]；具体内容参见 3.2.4 节。

3.2 一维数组

和简单变量一样，数组必须先定义，后使用。定义数组时，除了要给定数组的名称、数组成员类型，还要为其分配内存空间，并进行初始化。

3.2.1 定义数组

一维数组的定义格式如下：

```
数据类型 数组名[];
```

其中，数据类型是指数组成员的类型，可以为基本数据类型，也可以为引用数据类型。[]为数组标记。在定义数组时，不允许在[]内指定数组元素的个数。例如：

```
int a[],  boolean temp[],  String s[];
```

其中，数组 a 的数据类型为 int，temp 的数据类型为 boolean，s 的数据类型为类 String。

提示：定义数组时，另外一种可选形式是直接将方括号放在变量类型的后边。例如：

```
int[] a,  boolean[] temp,  String[] s
```

3.2.2 生成数组

数组定义只是建立了一种数组的引用，还必须使用关键字 new 为其分配内存空间，否则它是无法被访问的。基本形式如下：

```
数组变量名=new 数据类型[数组长度]
```

例如：

```
char s[];  s=new char[5];
```

其作用是为数组 s 分配 5×2 字节大小的空间。也可以在定义数组的同时为之分配内存空间，例如：

```
int temp[]=new int[10];
```

提示:必须给出数组长度,而且数组一旦创建,就不允许再增加它的空间。

3.2.3 初始化数组

当使用关键字 new 生成数组时,数组中的每个成员都会自动被初始化,初始值依据数组的类型而定。例如:

数值型的初始值:0;

字符型的初始值:'\0';

布尔型的初始值:false;

类对象的初始值:null。

也可以通过主动赋值的方式对数组进行初始化。例如:

```
int s[]=new int[3];        s[0]=1; s[1]=2; s[2]=3
```

或将数组的定义,内存空间分配,初始化工作放在一条语句中,格式如下:

```
数据类型 数组名 []={值 1,…,值 n}
```

这种数组初始化形式的特点是无须给出数组的长度,系统会初始值的数量自动确定数组长度,并依次将每个数组成员放在数组中。

例如:

```
int s[]={1,2,3};
```

使得数组长度为 3,且 s[0]=1,s[1]=2,s[2]=3。

数组不能整体赋值,例如下列语句是错误的:

```
char s[]=new char[5];     s[]={'a','b','c','d','e'};      //错误
```

3.2.4 访问数组

数组成员的访问是通过数组名和下标来实现的,Java 中下标是从 0 开始,依次递增 1,如果数组 temp 长度为 10,则数组下标从 0 到 9。这些成员分别表示为:

```
temp[0],temp[1],…,temp[9]
```

对数组中任意成员的引用形式为:

```
数组名 [下标]
```

例如,temp[4]表示引用数组 temp 的第 5 个元素。数组元素的用法和平常使用的普通变量没有什么差异,也可以被赋值或参与其类型允许的各种运算等。

例如:

```
int i=temp[0]%2;
temp[1]=temp[0] * 5;
System.out.println("value="+temp[2]);
```

使用数组时要注意数组是否越界,每个数组都有一个属性 length,可以通过数组名

.length来获取数组的长度,即数组元素的个数。

3.2.5　实用案例

【例3.2】 求一维数组的最大值及其所在位置。

为了给一维数组进行随机赋值,这里采用Math.random()方法产生随机数。

本实用案例程序如下:

```
package Code0302;

public class MaxNum {
    public static void main(String[] args) {
        final int ARRAY_SIZE=10;
        int a[]=new int[ARRAY_SIZE];
        int max=0;
        int index=0;                          //存储最大元素的位置
        for (int i=0; i<a.length; i++) {      //本例中 a.length=10
            a[i]=(int) (Math.random() * 10);  //产生随机数,并对数组成员赋值
            System.out.print("  "+a[i]);
        }
        System.out.println();
        max=a[0];
        for (int j=1; j<ARRAY_SIZE; j++) {
            if (a[j]>max)                     //判断当前位置的数是否大于最大值 max
            {
                max=a[j];
                index=j;                      //替换当前的最大值,记住对应位置
            }
        }
        System.out.println("A["+index+"] has maximum value "+a[index]);

    }
}
```

程序运行结果:

```
2  0  0  1  9  5  1  4  2  7
A[4] has maximum value 9
```

分析:第一个for循环依次将随机数读入下标为i的数组中,完成初始化操作;第二个for循环依次取出每个数组元素a[j]的值,并判断其是否为当前最大值。

提示:如果例3.2中第二个for循环改为for(int j=1;j<=ARRAY_SIZE;j++),则会出现数组下标越界的情况,因为a[10]并不存在。

3.3　二维数组

到目前为止使用的都是一维数组,但有时应用中需要使用多维数组才能充分存储一些数据。例如,(x,y)坐标系统中,如果需要存储坐标x和坐标y来标识一个点,则需要使用二

维数组,其中一维用于存储坐标 x,另一维用于存储坐标 y。

将数组的下标数称为数组的维数,维数大于 1 的数组称为多维数组。因为多维数组的操作比较相似,所以本节重点介绍二维数组。

3.3.1 定义二维数组

二维数组的定义与一维数组相似:

```
数据类型  数组名[][];
```

其中的两个括号表示这是一个二维数组。当然,它也支持另一种形式:

```
数据类型[][] 数组名;
```

和一维数组相似,二维数组的创建,可以先定义数组再分配内存空间,也可以同时进行这两项工作。例如:

```
int temp[][];  temp=new int[2][3];
int temp[][]=new int[2][3];
```

这样就创建了一个二维结构的数组,其中行数为 2,列数为 3。也可将其看作是一个长度为 2 的一维数组,每个数组成员又是一个长度为 3 的一维数组。这些数组成员按行排列如下:

```
temp[0][0], temp[0][1], temp[0][2]
```

或者

```
temp[1][0], temp[1][1], temp[1][2]
```

同样,可以在定义数组的同时对其进行初始化。例如:

```
int temp[][]={{1,2},{3,4},{5,6}}
```

系统自动根据初始值的情况,为数组指定大小长度,并依次将值填入相应的单元中。例如初始化后 temp[0][1]=2,temp[2][0]=5。

注意:等式右边大括号内嵌套的大括号不能省略,它代表数组的一行及其组成。

3.3.2 二维数组元素的引用

要引用二维数组元素,引用方式为:

```
数组名[下标 1][下标 2]
```

下标仍然是从 0 开始逐渐递增的,同样在引用时要注意数组的越界问题。二维数组也有 length 属性,可以求每一维数组的长度,例如:

```
int temp[][]=new int[3][5];
System.out.println(temp.length);        //求二维数组的长度实际是求它的行数 3
System.out.println(temp[0].length);     //每个数组成员又是一个一维数组,其长度为 5
```

【例 3.3】 求二维数组中各元素的和。

```
package code0303;
public class SumAll{
    public static void main(String args[]){
    int total=0;
    int arr[][]=new int[3][4];
    for(int i=0;i<arr.length;i++){          //初始化并显示二维数组
        for(int j=0;j<arr[i].length;j++){
            arr[i][j]=i+j;
            System.out.print(" "+arr[i][j]);
        }
        System.out.println();
    }
    for(int i=0;i<arr.length;i++)           //求和
        for(int j=0;j<arr[i].length;j++){
            total=total+arr[i][j];
        }
    System.out.println(" The Sum is:"+total);
  }
}
```

程序运行结果：

```
0 1 2 3
1 2 3 4
2 3 4 5
The Sum is:30
```

3.3.3　实用案例

【例 3.4】 求矩阵 **A** 与 **B** 的乘积。

根据矩阵的乘法规则，如 a[4，3]×b[3，2]将产生一个 r[4，2]的新矩阵，r[i][j]=a[i][0]×b[0][j]+a[i][1]×b[1][j]+a[i][2]×b[2][j]。

例如：

$$x=\begin{pmatrix}1 & 6\\3 & 8\end{pmatrix}\times\begin{pmatrix}2 & 2\\9 & 7\end{pmatrix}$$

其计算过程为：

$$1\times2+6\times9=56$$
$$1\times2+6\times7=44$$
$$3\times2+8\times9=78$$
$$3\times2+8\times7=62$$

因此结果为：$\begin{pmatrix}56 & 44\\78 & 62\end{pmatrix}$。

本实用案例程序如下：

```
public class MatrixMultiply {
    public void multiply(int[][] a,int[][] b) {
```

```
        int [][] r=new int[4][2];            //r用于存放运算结果
        int tmp=0;
        for (int k=0; k<r[0].length; k++) {
            //双重循环,遍历 a 矩阵
            for (int i=0; i<a.length; i++) {
                tmp=0;
                for (int j=0; j<a[0].length; j++) {
                    tmp+=a[i][j] * b[j][k];
                }
                r[i][k]=tmp;
            }
        }
         for (int i=0; i<r.length; i++) {
            for (int j=0; j<r[0].length; j++) {
                System.out.print(r[i][j]+"\t");
            }
            System.out.println();

        }
    }
    public static void main(String[] args) {
          int[][] a=new int[][] {
                        { 1, 2, 3 },
                        { 4, 5, 6 },
                        { 7, 8, 9 },
                        { 11, 12, 13} };
          int[][] b=new int[][]{
                        {1,2},
                        {3,4},
                        {5,6} };
          MatrixMultiply ma=new MatrixMultiply();
          ma.multiply(a,b);
    }
}
```

3.4 Arrays 类

3.4.1 Arrays 类及基本使用

为了更方便地操作数组,Java 在包 java.util 定义了一个称为 Arrays 的类,该类包含了如下几个用 static 修饰的静态方法。

(1) static type[] copyOf(type[] original,int length):将 original 数组复制为一个新数组,其中 length 为新数组的长度。

(2) static int binarySearch(type[] a,type key):使用二分搜索法在类型为 type 的数

组中搜索类型为 type 的指定值 key。

(3) static boolean equals(type[] a,type[] b)：比较两个类型为 type 的数组是否相等。

(4) static void fill(type[] a, type val)：用一个指定的值 val 填充类型为 type 的数组 a。

(5) static void fill(type[] a, int fromIndex, int toIndex, type val)：与前一个方法类似,但填充时仅仅针对下标为 fromIndex 到 toIndex－1 的数组元素赋值为 val。

(6) static void sort(type[] a)：对类型为 type 的数组排序。

每个方法的具体描述,请参考 JDK 文档。

【例 3.5】 Arrays 类的基本使用。

```
package code0304;

import java.util.Arrays;
public class ArraysDemo {
    public static void main(String[] args) {
        Integer array[]=new Integer[9];
        for (int i=1; i<10; i++)
            array[i-1]=(int) (Math.random() * 100);
        //显示,排序数组
        System.out.print("原内容:");
        display(array);
        Arrays.sort(array);
        System.out.print("排序后:");
        display(array);
        //将值-1分配给数组 array 中下标从 0到 3-1位置上的元素
        Arrays.fill(array, 0, 3, -1);
        System.out.print("执行 fill()后:");
        display(array);
        //搜索 39
        System.out.print("值 39 的位置 ");
        int index=Arrays.binarySearch(array, 39);
        System.out.println(index);
    }
    static void display(Integer array[]) {
        for (int i=0; i<array.length; i++)
            System.out.print(array[i]+" ");
        System.out.println("");
    }
}
```

程序运行结果:

```
原内容:90 48 81 14 3 35 95 4 97
排序后:3 4 14 35 48 81 90 95 97
执行 fill()后:-1 -1 -1 35 48 81 90 95 97
值 39 的位置 -5
```

提示:Java 8 对 Arrays 类的功能进行了增强。例如增加了一些新的方法可以利用多 CPU 的并行处理能力提高数组排序、赋值等操作性能,如 parallelSort(type[] a)方法与之前介绍的 sort()方法相似,但可以在多 CPU 上并行实现。

3.4.2 实用案例

【例 3.6】 对数组按中文名称排序。

本实用案例程序如下:

```
package code0304;

import java.text.Collator;
import java.util.Arrays;
import java.util.Comparator;
public class SortByChinese {
    public static void main(String[] args) {
        String[] arrStrings= {"计算机","长江", "通信","数学"};
        Arrays.sort(arrStrings);
        for (int i=0; i<arrStrings.length; i++)
            System.out.println(arrStrings[i]);
        System.out.println("----------------------------");
        //Collator 类是用来执行区分语言环境的字符串比较,这里选择用 CHINA
        Comparator comparator=Collator.getInstance(java.util.Locale.CHINA);
        //根据指定比较器产生的顺序对指定对象数组进行排序
        Arrays.sort(arrStrings, comparator);
        for (int i=0; i<arrStrings.length; i++)
            System.out.println(arrStrings[i]);
    }
}
```

程序运行结果:

```
数学
计算机
通信
长江
----------------------------
长江
计算机
数学
通信
```

Arrays 类中的 sort()方法缺失一般是按照数组中数值的大小或字母顺序进行排序的,但这种处理方式对中文是无效的,如上例运行结果。为此使用了类 Arrays 中另一种形式的 sort()方法:sort(T[] a,Comparator<?super T>c),它可以根据指定比较器(Comparator)产生的顺序对对象数组进行排序。为获取 Comparator 对象,可以通过方法 Collator.getInstance()实现,其中参数 java.util.Locale.CHINA 表示按中文语言进行排序。

3.5　数组实训任务

【任务描述】

编写一个模拟的Java发牌程序，要求将两副牌，也就是108张，发给4个人，并留8张底牌，最后输出底牌和每个人手中的牌。

【任务分析】

(1) 首先确定扑克牌在计算机中的表达方式，因为计算机中无法表达扑克牌中的花色，因此最好为每张牌设定一个编号，拟定的编号规则如下：

① 红桃按照从小到大依次为：1～13。

② 方块按照从小到大依次为：14～26。

③ 黑桃按照从小到大依次为：27～39。

④ 梅花按照从小到大依次为：40～52。

⑤ 小王为53，大王为54。

(2) 由于可发的牌和每个玩家手中的牌都需要记录，且其由多个有序数据组成，因此需要设计两个数组来存放108张牌和每个玩家手中的牌。

(3) 整个程序由以下4部分构成：

① 按照以上编号规则初始化一个包含108个数字的数组。

② 每次随机从该数组中抽取一个数字，分配给保存玩家数据的数组。

③ 循环输出每个玩家手中的牌。

④ 最后输出底牌。

【任务解决】

完整的示例程序如下：

```
package code0305;

import java.util.*;
public class CardPlay{
    public static void main(String[] args){
        int[] total=new int[108];           //存储108张牌的数组,定义方法见3.2.1节
        int[][] player=new int[4][25];      //存储4个玩家的牌,定义方法见3.3.1节
        int leftNum=108;                    //当前剩余牌的数量
        int ranNumber;
        Random random=new Random();         //生成Random对象,用以生成随机数
        for(int i=0;i<total.length;i++){    //初始化一维数组
            total[i]=(i+1)%54;
            if(total[i]==0){                //处理大小王编号
                total[i]=54;
            }
        }
//循环发牌
        for(int i=0;i<25;i++){                          //为每个人发牌
            for(int j=0;j<player.length;j++){           //生成随机下标
```

```
                    ranNumber=random.nextInt(leftNum);           //random.nextInt()方法生成随机数
                    player[j][i]=total[ranNumber];         //发牌
                    total[ranNumber]=total[leftNum-1];     //删除已经发过的牌
                    leftNum--;
                }
            }
    //循环输出玩家手中的牌
            for(int i=0;i<player.length;i++){           //通过两层循环遍历二维数组,见3.3.2节
                System.out.print("玩家"+i+"的牌:");
                for(int j=0;j<player[i].length;j++){
                    System.out.print(" "+player[i][j]);
                }
                System.out.println();
            }
    //底牌
            System.out.print("底牌:");
            for(int i=0;i<8;i++){
                System.out.print(" "+total[i]);
            }
            System.out.println();
        }
    }
```

程序运行结果:

```
玩家 0的牌: 52 15 12 53 32 24 48 36 24 11 54 27 14 22 3 8 10 40 43 46 40 45 37 41 22
玩家 1的牌: 49 25 5 47 20 35 45 51 29 17 10 26 20 51 3 1 29 43 31 9 33 30 32 4 53
玩家 2的牌: 42 7 13 38 34 50 12 6 39 49 33 23 14 31 7 19 39 48 16 18 35 5 41 9 11
玩家 3的牌: 44 2 30 26 28 46 44 19 21 23 25 21 15 13 1 28 34 52 4 8 6 47 17 2 27
底牌: 50 42 18 37 16 38 36 54
```

习题与思考

1. 为了定义3个整型数组a1、a2、a3,下面声明正确的语句是(　　)。
   A. intArray [] a1,a2;　int a3[]={1,2,3,4,5};
   B. int [] a1,a2;　int a3[]={1,2,3,4,5};
   C. int a1,a2[];　int a3={1,2,3,4,5};
   D. int [] a1,a2;　int a3=(1,2,3,4,5);
2. 编程实现任意两个数组的相乘。
3. 将给定数组 int a[]={78,23,56,34,12,45,67,89}按从小到大进行排序并输出。
4. 给出下列程序段运行的结果:

```
public static void main(String args[])
{
```

```
        int array[]={1,2,3,4,5};
        printarray(array);
        could_modify(array);
        printarray(array);
    }
    static void could_modify(int a[]){
        for(int i=0;i<a.length;i++)
            a[i] *=i;
    }
    static void printarray(int a[])
    {
        for(int i=0;i<a.length();i++)
        System.out.println(a[i]);
    }
```

5. 数组创建后,可否通过设置变量 length 的大小来增减数组的大小?

第 4 章 类和对象设计

本章学习目标

Java 程序一般由一个主类和支持这个主类的其他相关类构成，之前已经初步接触和使用到 Java 类库中的一些基本类（如 String 等），本章将学习建立属于自己的类，并通过其构建实用的应用程序。本章主要内容包括：

（1）类的定义。

（2）对象的创建和使用。

（3）构造方法的定义和使用。

（4）类的成员变量与成员方法。

（5）方法的重载。

（6）权限修饰符。

4.1 面向对象基础

要设计一个计算机程序，一种方法是根据问题解决的步骤，将程序组织为一系列的线性代码或函数，然后依次调用或访问即可，如 C 或 Pascal 程序设计语言。

面向对象程序设计（Object Oriented Programming，OOP）语言则是从另一个角度看待计算机程序，其核心是引入了对象的概念，使之可以逼真地模拟现实世界中的任何事物，从而达到与人类的思维习惯相一致。

现实世界中存在很多不同的对象，如人、车、家用电器等，每个对象都有自己独特的属性和行为，可以提供不同的功能和服务，一方面对象之间各自独立，另一方面它们又相互联系，通过协作的方式完成复杂的任务。例如，“扬声器”可以播放中频和高频声音，“喇叭”可以播放低音，“高频头”可以接收无线电广播，“CD 播放器”可以读取光盘音频数据。这些对象似乎彼此独立，但一旦将其连接起来，安装上驱动程序，则可以构成一个复杂的音响系统，播放各种音乐。

面向对象程序设计充分借鉴这一思想，将现实世界中的事物及其关系映射到问题空间中，设计多个对象并使其各司其职，分别完成一组相关的任务。如果一个对象依赖于一个不在其控制范围内的方法，它就需要访问包含该方法的其他对象，即请求其他对象提供该方法

或服务，借用 OOP 的术语来说，称为一个对象向另一个对象发送消息。

例如“CD 播放”任务的实现，需要“CD 播放器”和“扬声器”两个部件的同时参与才能完成，因此，可以对应设计两个对象，分别实现“读取数据”和“发声”的功能，并将其组合起来，以完成上述功能。

其实，这是面向对象程序设计中一种典型的设计思想——“抽象”：将现实世界中的实体理解为由属性数据和对这些属性实施行为的统一体，即对象。属性表示对象的性质，行为则定义了对象可以提供的外部服务。例如，将“CD 播放器”视为对象，则它具有播放速度、重量、形状等属性，对外可提供读数据、写数据等操作，并且这些操作将部分影响或改变“CD 播放器”的属性。面向对象程序设计中其他三个典型特征是：封装、继承和多态。

1. 封装

封装是一种将操作和操作所涉及的数据捆绑在一起，使其免受外界干扰和误用的机制。例如将录音机视为一个对象，如图 4-1 所示。

它的属性包括运转方向、运转速度等，提供的行为包括 Play（播放）、Back（后退）、Record（录音）和 Forward（前进）等，使其围绕在属性数据周围。这样所有属性均无法直接访问，因为它们被上述方法封装保护起来，使用者只能通过提供的方法改变或访问属性的值，就像只能通过录音机的按键来使用它，而不是直接改变其速度、方向一样。

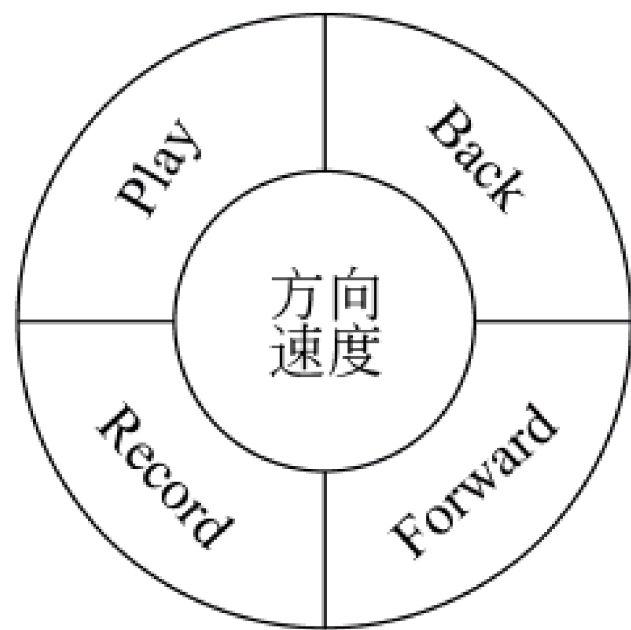


图 4-1　录音机中的封装

在 Java 中，最基本的封装单元是类，一个类定义为具有相似特征（属性和行为）对象的一种抽象。

2. 继承

继承是指一个新的类继承原有类的基本特征，并可增加新的特性。原有的类称为父类或基类，新的类称为子类或派生类。在子类中，不仅包含父类的所有数据和方法，还可增加新的属性和方法。继承具有传递性。如果 B 继承自 A，C 继承自 B，则 C 间接继承自 A，这样就形成了一个层次关系，如图 4-2 所示。

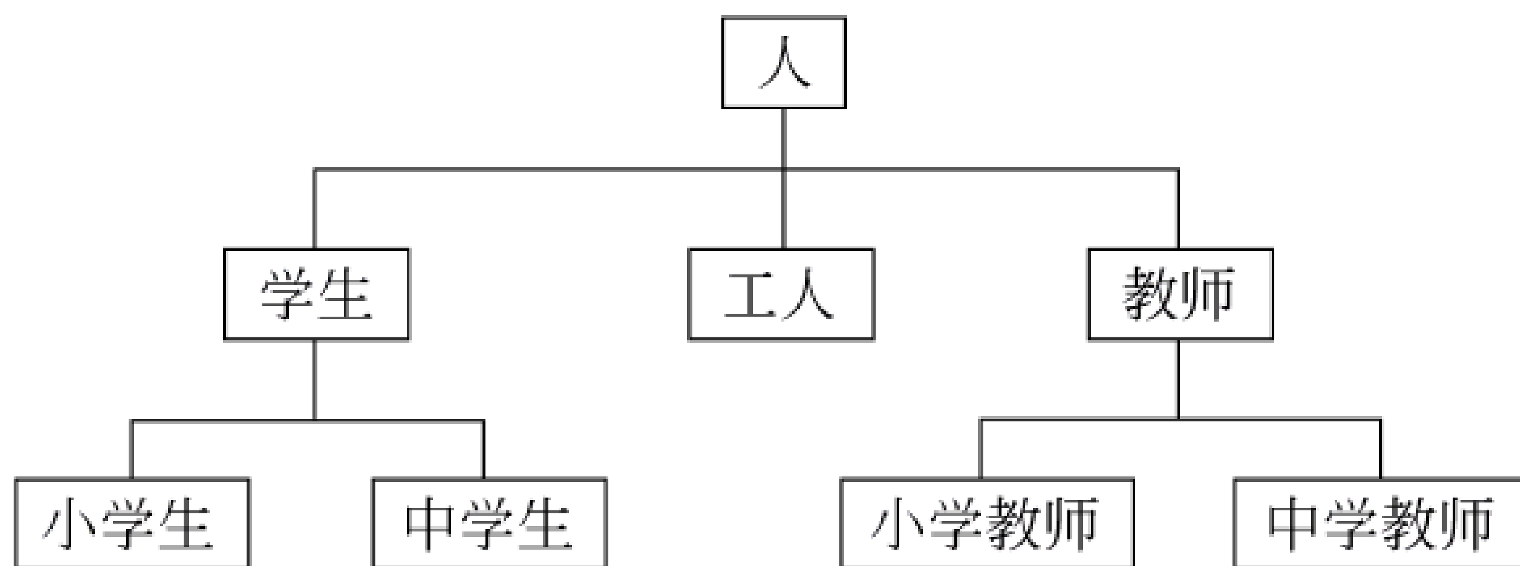


图 4-2　人们间的继承关系

根据类的继承机制，在父类中只定义各子类共同需要的属性和方法。类派生时，可以增加新的属性和方法。因此，父类的基本特征可被所有子类对象共享，提高了类的重复利用率。

3. 多态性

多态性是指类中同一名称的行为（方法）可以有多种不同的功能，或者相同的接口有多种实现方法，具体内容参见第 5 章。

4.2 类和对象初探

类是 Java 中最重要的复合数据类型,是组成 Java 程序的基本元素,它封装了一类对象的属性和改变这些属性的方法,是这一类对象的原型。与简单数据类型的使用不同,用户必须先定义类,并生成该类的实例,然后才能通过该实例访问其成员变量和方法。

例如,圆可以有很多种,但所有的圆均由圆心位置和半径决定,其基本行为至少包括计算面积和周长等。通过抽象,可以定义类 Circle。

【例 4.1】 定义类 Circle,并输出任意圆的面积和周长。

```
package code0402;

class Circle {
    float x, y;                              //定义圆心 x,y坐标
    float radius;                            //定义成员变量(半径)
    double getArea() {                       //定义计算面积的方法
        return radius * radius * Math.PI;
    }
    double getCircumference() {              //定义计算周长的方法
        return 2 * radius * Math.PI;
    }
    public static void main(String args[]) {
        Circle c=new Circle();               //创建类 Circle的对象 c
        c.radius=3;
        c.x=0;
        c.y=0;                               //为对象 c中变量赋值
        System.out.println("Area="+c.getArea());          //访问成员方法
        System.out.println("Circumference="+c.getCircumference());
    }
}
```

程序运行结果:

```
Area=28.27
Circumference=18.84
```

分析:整个程序可分为以下 3 个部分。

(1) 通过关键字 class 定义类 Circle,包括圆心、半径属性以及周长、面积的计算方法。

(2) 在主函数 main 中通过语句 Circle c=new Circle()生成类 Circle 的对象 c。

(3) 通过点运算符“.”访问对象 c 的成员变量和方法。

从对象的整个生命周期来看,它包含了类的定义(4.3 节)、对象实例化(4.4.1 节)、对象初始化(4.4.2 节)、对象使用(4.4.3 节)和对象清除等若干个阶段,下面逐一对这些阶段的工作进行分析。

4.3 定 义 类

类通过关键字 class 进行标识，基本形式如下：

```
[类修饰符] class 类名 [extends 父类名] [implements 接口名]
{
…//类体,包括定义类的成员变量和方法
}
```

类名由用户指定，可以是任意合法的标识符。

类体是定义在大括号中的部分，它是整个类的核心，可以分为类的成员变量和方法两个部分。

类修饰符为可选项，它决定了类在程序运行过程中以何种方式被处理。定义类时，可以接受默认的修饰符，或其他类型修饰符（如 abstract、final 等）。

“extends 父类名”也为可选项，表示所定义的类继承自其他父类，这时该类自动获得父类中所有可能的属性和方法，并可添加父类没有的其他成员。

“implements 接口名”也为可选项，它表示定义的类需要通过实现某个接口完成，接口实际也是一种特殊的类，它所定义的方法一般为空，需要在派生的类中实现该方法。类的继承和实现将在第 5 章中介绍。

4.3.1 定义成员变量

成员变量的定义格式为：

```
[修饰符]类型  成员变量名列表;
```

成员变量的类型可以为任意数据类型，包括简单类型（如整型、浮点型）、数组、类（如 String）等，在一个类中，成员变量名是唯一的。类的成员变量定义在所有的方法体之外，因此它的作用域是整个类，即从变量定义开始一直到标识类体结束的}处。

类中的所有方法均可以直接访问成员变量。

成员变量的修饰符（如 public、protected、private、final 和 static 等）为可选项，主要起到对变量进行访问控制和限定的作用，这些将在 4.6 节和 5.4 节具体讨论。

【例 4.2】 定义学生类，其中含有属性学号、姓名、出生日期等，其中出生日期又含有属性年、月、日等。

```
class Student{
  int no;
  String name;
  Birthday day;          //day的类型为类 Birthday
}
class Birthday{          //定义类 Birthday
  int year;
  int month;
  int date;
}
```

4.3.2 定义成员方法

方法表示对象所具有的行为,其基本定义格式如下:

```
[修饰符] 返回值类型 方法名 ([参数列表])
{
    方法体
}
```

返回值类型可以是任意基本类型或类,如果方法不返回任何值,它必须声明为 void (空)。参数列表由零个或多个参数构成,参数之间用逗号分隔,每个参数由一个数据类型和一个标识符构成,通过参数列表可以将参数值传递给被调方法。

方法体部分定义了该方法是如何实现的,它由{和}内的语句序列组成。方法如果有返回值,可通过 return 语句传递给调用者,且必须和定义的返回值类型一致。

【例 4.3】 定义二维空间中的点 Point 和基本方法。

```
package code0403;

class Point {
    int x=0;
    int y=0;
    public void move(int dx, int dy)        //移动点坐标的方法
    {
        x=x+dx;
        y=y+dy;
    }
    public void alert() {                   //无返回值的输出方法
        System.out.println("x="+x+" y="+y);
    }
    public static void main(String args[]) {
        Point p=new Point();
        p.move(1, 2);
        p.alert();
    }
}
```

程序运行结果:

```
x=1  y=2
```

提示:方法体中的变量均为局部变量,参数列表中的变量也是局部变量,因此只能在本方法的{…}内使用,如方法 move 中的局部变量 dx、dy 不能在方法 alert 中使用。但类中成员变量的作用域是整个类,因此变量 x、y 可以在方法 alert 中使用。

Java 中有两种特殊的成员方法:构造方法和 main()方法,其定义形式与传统方法稍有不同:

构造方法要求方法名必须与类名相同,且不能有返回值(void 也不行)。例如,类

Student 的构造方法可定义为：

```
Student(int i_no, String i_name)
```

或

```
Student(String s)
```

main()方法通常是一个 Java 应用程序的执行起点，其定义格式为：

```
public static void main(String args[])
```

其中，main 为方法名，args 为形参，类型为字符串数组，void 表示 main 方法无返回值。

4.3.3 方法重载

Java 允许使用同一个名字去定义多个方法，只要方法的参数列表不同，即参数的数量、类型不完全相同。方法调用时，编译器根据实参列表的个数和类型自动调用匹配的方法。

方法重载使得类中两个相似的方法可以拥有完全相同的名字，也可以更灵活的基于所接收的参数调用不同的方法。

【例 4.4】 编程实现对双操作数和三操作数加法运算的重载。

```
package code0403;
public class OverLoad {
    int sum(int a, int b) {                    //定义双操作数的 sum 方法
        return a+b;
    }

    int sum(int a, int b, int c) {             //重载的三操作数 sum 方法
        return a+b+c;
    }

    public static void main(String[] args) {
        OverLoad o=new OverLoad();
        System.out.println(o.sum(1, 5));       //调用双操作数 sum 方法
        System.out.println(o.sum(3, 5, 8));    //调用三操作数 sum 方法
    }
}
```

程序运行结果：

```
6
16
```

分析： 类 OverLoad 对成员方法 sum()进行了重载，两个方法的参数个数不同，调用时 o.sum(1,5)调用第一个 sum()方法，o.sum(3,5,8)调用第二个 sum()方法。

提示： 根据方法的返回值类型是无法区分重载方法的，因为在调用方法时，返回值是不参与调用的。

4.4 对　　象

4.4.1 实例化对象

类作为一种抽象的复合数据类型,必须先要实例化(即生成对象),然后才能使用。基本语句格式为:

```
类名 对象名=new 类名 ([参数列表])
```

例如,实例化类 Point 的两个对象 p1、p2:

```
Point p1=new Point ();
Point p2=new Point ();
```

关键字 new 为每个生成的对象分配一片内存区域,并返回该对象的一个引用(可以理解为该对象的内存首地址)。

可以为一个类创建多个不同的对象,每个对象占用不同的内存空间,存储各自的状态信息,改变其中一个对象的状态不会影响其他对象,如图 4-3 所示。

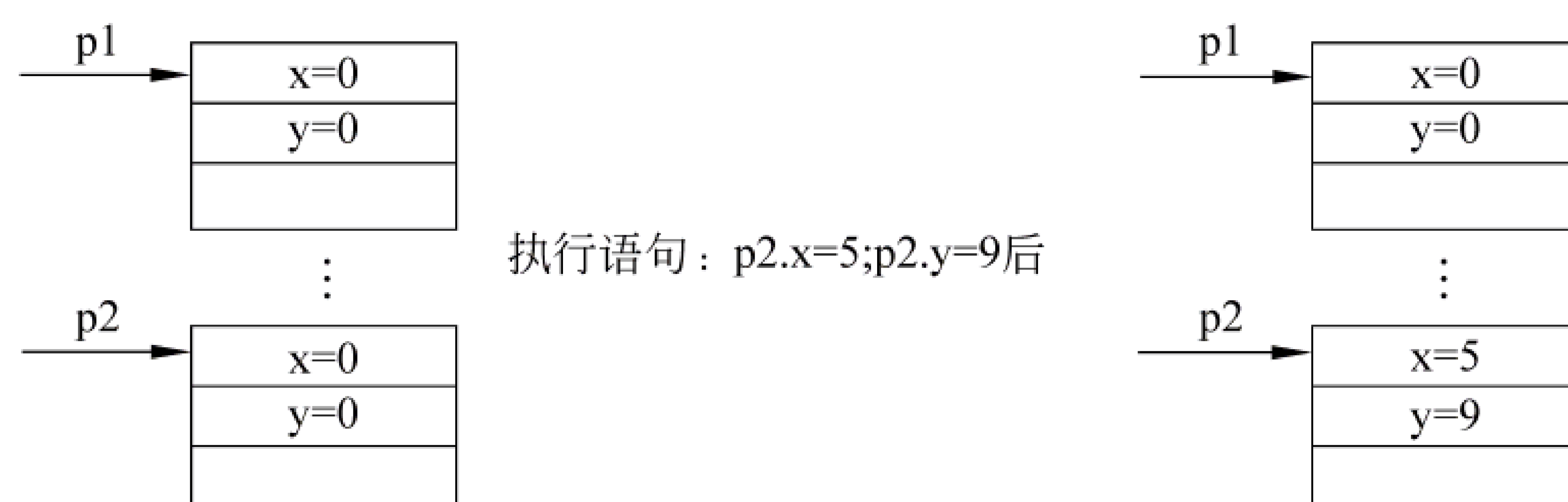


图 4-3　对象 p2 的属性变化与 p1 无关

提示:当运算符==和!=用于两个对象的比较时,它不是比较两个对象的"值"是否相等,而是判断运算符两边是否是同一个对象(地址空间相同),如上述对象 p1 和 p2 显然是两个不同的对象,因为它们的地址空间完全不同,因此表达式 p1==p2 的值为 false。如果要比较两个对象的"值"是否相等,可以使用 equals()方法,如 p1. equals(p2)。

如果只想声明某类的实例,而不想为其创建任何对象,则可以使用关键字 null(含义为空),例如:

```
Point p3=null;
```

这样就声明了类 Point 的对象 p3,但并没有为其分配内存空间,在需要的时候再通过语句 p3=new Point()对其实例化。

4.4.2 初始化对象

初始化对象就是为所创建对象的成员变量赋初值,其中最直接的方式就是在定义成员变量的同时对其赋值。例如:

```
class Student{
```

```
    String name="ZhangShan";
    …
}
```

这样,每个 Student 对象中变量 name 的值均为 ZhangShan。

一种更好的方式是使用 4.3 节中提到的构造方法,在为对象分配内存空间的同时,实现对象的初始化。具体这项工作包括两个子任务:

(1) 定义一个或多个构造方法,并在方法体中对成员变量赋初值。

(2) 调用或执行相应的构造方法。

【例 4.5】 通过构造方法对 Student 对象进行初始化。

```
package code0404;

class Student {
    int no;
    String name;

    Student(int l_no, String l_name) {              //定义构造方法
        this.no=l_no;
        this.name=l_name;                           //对成员变量 no,name 初始化
    }

    public static void main(String args[]) {
        Student s1=new Student(1, "ZhangShan");     //l_no=1;l_name=ZhangShan
        Student s2=new Student(2, "XiaoMing");      //l_no=2;l_name=XiaoMing
        System.out.println("name="+s1.name+" no="+s1.no);
        System.out.println("name="+s2.name+" no="+s2.no);
    }
}
```

程序运行结果:

```
name=ZhanShan   no=1
name=XiaoMing   no=2
```

需要说明的是:构造方法的调用方式和过程非常特别,它不是通过显示指定方法名直接调用的,而是在通过 new 实例化对象时由系统自动调用的。实际上,当新建一个对象时,系统将自动完成以下 3 项工作:

(1) 为每个对象分配不同的内存空间。

(2) 如果类定义时有初值,使用该值对成员变量进行初始化;如果没有初值,可以使用默认值。

(3) 自动调用构造方法,如果构造方法有多个,则根据参数的类型、个数等决定调用哪种构造方法。

由此可知,当语句 Student s1=new Student(1,"ZhanShan")被执行以创建对象 s1 时,构造方法同时被调用,并通过参数传递使得变量 no=1; name=ZhangShan。

如果用户在类中没有定义任何构造方法,则系统会自动生成一个默认的构造方法。该

方法没有参数,且方法体为空。例如,类 Student 的默认构造方法为:

```
public Student(){ }
```

当对该类进行实例化且不指定任何参数时,例如执行语句 Student s1=new Student(),系统将自动调用该默认的构造方法。

与常规方法相同,构造方法也可以被重载,以提供对象不同的初始化过程和属性值。

【例 4.6】 构造方法重载示例。

```
package code0404;

public class StudentOverload {
    int no;
    String name;
    StudentOverload(String l_name) {
        no= 0;
        name= l_name;
    }
    StudentOverload(int l_no, String l_name) {
        no= l_no;
        name= l_name;
    }
    public static void main(String args[]) {
        StudentOverload s1= new StudentOverload("ZhangShan");
        StudentOverload s2= new StudentOverload(2, "XiaoMing");
        System.out.println("name= "+ s1.name+ " no= "+ s1.no);
        System.out.println("name= "+ s2.name+ " no= "+ s2.no);
    }
}
```

程序运行结果:

```
name= ZhangShan no= 0
name= XiaoMing no= 2
```

分析:当程序中有多个重载的构造方法时,Java 根据 new 语句中的实参个数、类型自动调用与之匹配的构造方法。例如,语句 new Student("ZhangShan")中只有一种 String 类型的实参,因此应该匹配第一个构造方法;而语句 new Student(2,"XiaoMing")中第一个实参为整数,第二种实参为 String,与之匹配的是第二个构造方法。

4.4.3 使用对象

Java 中对象的使用包括对成员变量的引用和成员方法的调用,它们都是通过点运算符(.)来实现的,格式如下:

```
对象名.成员变量名
对象名.成员方法名(实参列表)
```

方法调用时，要指定被调用的方法名称、实际参数，并且要求实参的类型、个数和顺序与定义中的形参列表一致。下面通过例子加以说明。

【例 4.7】 对象的基本使用方法示例。

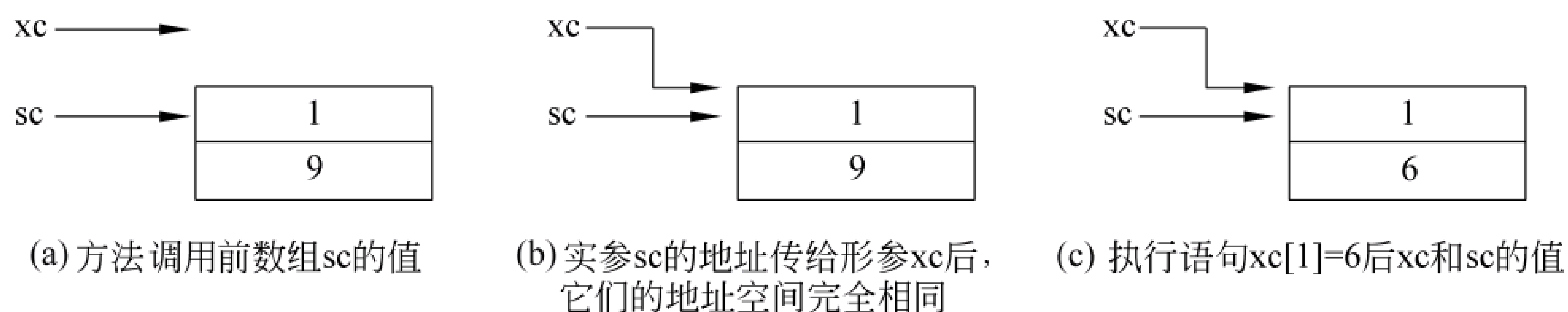
```
package code0404;
public class PassTest {
    float ptValue;
    public void changeInt(int value) {
        System.out.println("In changeInt:"+ value);
        value= 55;
    }
    public void changeArrayValue(int xc[]) {
        System.out.println("In changeArrayValue:"+ xc[1]);
        xc[1]= 6;
    }
    public static void main(String args[]) {
        int val= 11;
        int sc[]= { 1, 9 };
        PassTest pt= new PassTest();
        pt.ptValue= 20;                  //引用成员变量 ptValue
        pt.changeInt(val);               //方法调用 (传值形式)
        System.out.println("Current Int value is: "+ val);
        pt.changeArrayValue(sc);         //方法调用 (引用形式)
        System.out.println(" Current Value in array is: "+ sc[1]);
    }
}
```

程序运行结果：

```
In changeInt:11
Current Int value is: 11
In changeArrayValue:9
Current Value in array is: 6
```

分析：本例使用了两种不同的方法调用形式：传值调用和引用。传值调用的特征是形参一般为基本类型的变量(如数值型、字符型和布尔型等)。调用时，实参的值对应传递给形参，这种传递是单向的，即形参的值不会改变实参的值。例如上例中，尽管方法 changeInt()对形参 value 的值作了改变，但这种改变并不影响实参 val。

引用则类似于 C 语言中的传地址调用，直接将实参的地址传递给形参，这样两者使用完全相同的地址空间，因此实参可以改变形参的值，形参也可以改变实参的值。当实参为复合数据类型时(如数组、对象)，Java 自动采用引用方式进行参数传递。如例 4.7 中，方法 changeArrayValue()中传入的参数为数组，这时实际传递的是实参数组的引用，如图 4-4 所示，对形参的改变同时也意味着对实参的改变。



(a) 方法调用前数组sc的值　(b) 实参sc的地址传给形参xc后，它们的地址空间完全相同　(c) 执行语句xc[1]=6后xc和sc的值

图 4-4　基于引用的方法调用过程示例

4.4.4　使用静态变量和方法

对不同的对象而言，它们在内存中的存储空间通常是独立的，对某个对象成员变量值的修改不会影响到其他的对象，把这种只属于单个对象的成员变量称为实例变量。

如果所有对象希望共享同一变量，则可以在变量定义前加上一个关键字 static，这样就定义了一个类变量，也称静态变量。静态变量不再属于某个实例或对象，而是属于整个类。

类似的，可以通过在方法定义前加上关键字 static，将该方法定义为类方法或静态方法。例如：

```
static int no;                    //定义静态变量 no
static void alert(){ … }          //定义静态方法 alert()
```

由于静态变量和静态方法都属于整个类，因此可以直接通过类名访问，而无须创建该类的实例，基本格式如下：

```
类名.静态变量名;
类名.静态方法名([实参列表])
```

例如，重新定义类 StudentStatic 如下：

```
class StudentStatic{
    static int no;
    static String name="test";
    static void alert(){ System.out.println(name); }

}
public class SimpleTest{
    public static void main(String args[]){
        System.out.println(StudentStatic.no);    //通过类名引用静态变量 no
        StudentStatic.alert();                   //通过类名引用静态方法 alert()
    }
}
```

注意：对实例成员变量，每个实例都有一个；而对于静态成员变量，每个类只有一个，该类对应的所有实例均共享该变量，它们的区别可通过下面的例子说明。

【例 4.8】　实例变量和静态变量的区别示例。

```
package code0404;
```

```
class StaticTest {
  static int statInt=4;
  static double statDouble=16.0;
  int instInt;
  double instDouble;
  public static void staticMethod(){                //输出静态变量的值
     System.out.println ("statInt="+statInt+"; statdouble="+statDouble);
  }
  public void instMethod(){                         //输出实例变量的值
     System.out.println("instInt="+instInt+"; instdouble="+instDouble);
  }
  public StaticTest(int intArg, double doubleArg){
       instInt=intArg;
       instDouble=doubleArg;
  }
  public static void changestatic(int newInt, double newDouble){
                                                    //改变静态变量的值
    statInt=newInt;
    statDouble=newDouble;
  }
  public static void main(String args[]){
       StaticTest instance1=new StaticTest(1,2.0);
       StaticTest instance2=new StaticTest(3,4.0);
       instance1.instMethod();                      }//调用实例方法
       instance2.instMethod();
       StaticTest.staticMethod();
       instance1.staticMethod();                    }//调用静态方法
       instance2.staticMethod();
       instance1.changestatic(8,8.0);               //改变静态变量的值
       instance2.staticMethod();                    }//再次输出静态变量的值
       StaticTest.staticMethod();
    }
}
```

程序运行结果：

```
instInt=1; instdouble=2.0
instInt=3; instdouble=4.0
statInt=4; statdouble=16.0
statInt=4; statdouble=16.0
statInt=4; statdouble=16.0
statInt=8; statdouble=8.0
statInt=8; statdouble=8.0
```

程序分析：

(1) 对静态变量和静态方法，可以通过“类名. * ”和“对象名. * ”两种方式引用，结

果完全相同。

(2) 对实例变量,对象 instance1 和 instance2 提供了不同的内存空间,因此即便同名,也是两个不同的变量,其值各不相同。

(3) 静态变量 statInt 和 statDouble 对类 StaticTest 的所有对象都是公共的,因此通过 instance1 对其改变后,通过 instance2 再去访问时其值已经变化了,这进一步证明静态变量不是属于某个对象,而是属于整个类。

4.4.5 清除对象

对象使用完后,如果老占着内存不放,就会很快耗尽内存资源,因此必须及时清理。C 和 C++ 中的垃圾清理是由程序员负责完成的,有时,这是一件很困难的事情。因为用户并不总是事先知道垃圾应在何时被释放。Java 语言解除了程序员进行垃圾清理的责任,它提供一种系统级的线程跟踪每一块内存的分配情况。Java 虚拟机检测何时一个对象不再有用,然后收回该对象的空间为新对象所用。但是,如果你的对象用到任何其他系统资源,如在某个对象的生存期内打开了一些文件,而当这个对象被破坏时,若想确认这些文件是否已经被正确地关闭,那么可以重写 finalize()方法,它的格式如下:

```
protected void finalize() throws Throwable{
    …  //撤销对象
}
```

4.4.6 应用程序与命令行参数

一个 Java 应用程序(application)通常由一个或多个类构成,但其中只能有一个类作为整个应用程序执行的起点,称之为主类。

主类的特征之一是类名与 Java 文件名相同,且必须含有 main()方法,此外方法必须被声明为 public、static、void,具体格式如下:

```
public static void main(String args[]){
  …  //方法体
}
```

public 指明该方法可以被所有类使用,static 则表明该方法是静态方法。

由于 main()方法是应用程序执行过程中第一个被调用的内容,因此可以在该方法体中启动应用程序的任何其他代码,包括生成其他类的实例等。

与其他方法类似,main()方法也可以接收一个或多个(命令行)参数,这些参数作为字符串依次被传递并存入数组 args[]中,即 args[0]存放第一个参数,args[1]存放第二个参数,以此类推,但要注意:Java 不把文件名作为参数之一。

【例 4.9】 命令行参数使用示例。

```
package code0404;

public class MainTest {
    public static void main(String args[]) {
        int n=args.length;          //取得参数的个数
```

```
        if(n==0)
            System.out.println(" no parament ! ");
        else {
            System.out.println(" number of paraments: "+n);
            for (int i=0; i<n; i++)
                System.out.println(" args["+i+" ]="+args[i]);
        }
    }
}
```

编译通过后，执行该程序：

```
java  MainTest  I  Love  China
```

或通过可视化开发工具配置，在 Eclipse 的 Package Explorer 中先找到对应类文件，右击，在弹出菜单上选择 Run as-Run Configurations 选项，在弹出窗口中选择 Arguments 选项卡，在 Program arguments 栏输入命令行参数：I Love China，然后单击 Run 按钮，如图 4-5 所示。

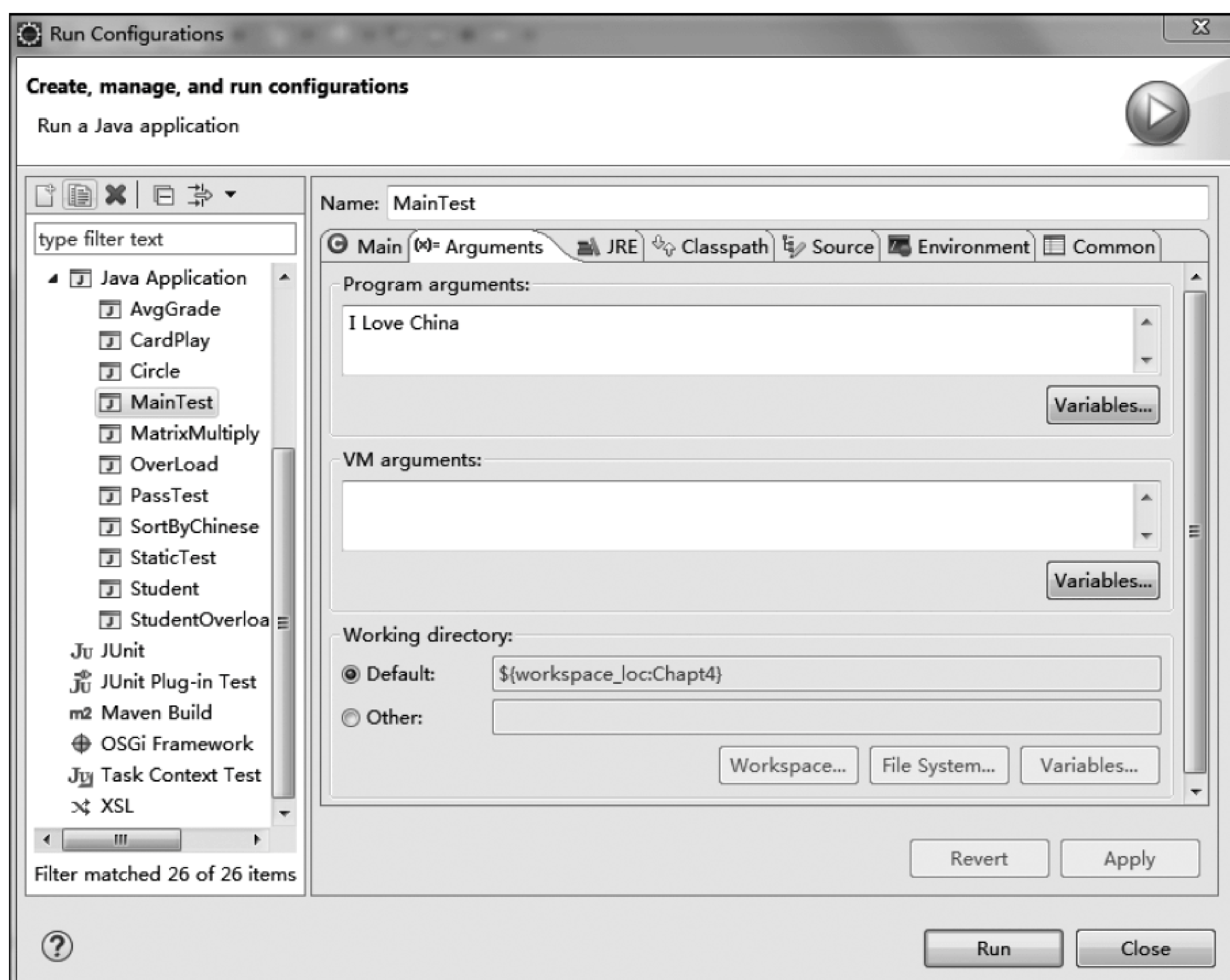


图 4-5　在 Eclipse 中使用命令行参数

程序运行结果：

```
number of paraments: 3
args[0 ]=I
args[1 ]=Love
args[2 ]=China
```

4.4.7 实用案例

【例 4.10】 实现一冒泡排序算法。

假定用户可以任意指定待排序的数字序列,可以利用 Scanner 类获取用户从控制台的输入,调用 Scanner 的 nextLine()方法,对指定的数字序列排序,详见第 8 章。

本实用案例程序如下:

```
package code0404;

import java.util.Scanner;
public class BubbleSort {
    public static void main(String args[]) {
        Scanner scanner=new Scanner(System.in);
                                        //Scanner 用于获取用户从控制台的输入
        int[] numbers=new int[8];
        System.out.print("Please enter eight number:");
        for (int i=0; i<numbers.length; i++)
            numbers[i]=scanner.nextInt();
                                        //通过 nextInt 方法依次读取用户输入数字并存入数组中
        for (int i=numbers.length-1; i>0; i--) {
            for (int j=0; j<i; j++) {
                if (numbers[j]>numbers[j+1]) {
                    int temp=numbers[j];
                    numbers[j]=numbers[j+1];
                    numbers[j+1]=temp;
                }
            }
        }
        System.out.println("冒泡排序的结果是:");
        for (int i=0; i<numbers.length; i++) {
            System.out.print(numbers[i]+" ");
        }
    }
}
```

程序运行结果:

```
Please enter eight number:   24   18   36   30   7   15   9   1
```

冒泡排序的结果是:

```
1 7 9 15 18 24 30 36
```

【例 4.11】 实现一对象计数器。

当新创建一个该类的对象时,计数器自动加 1,当删除一个该类的对象时,计数器自动减 1。

提示:可以借助静态成员变量实现。

本实用案例程序如下：

```
package code0404;

public class Counter {
    public static int num=0;            //用于记录对象数的静态成员变量
     public Counter() {
         num++;
         System.out.println("after new NO="+num);
     }

     public static void delete(Counter o) {
        num--;
        System.out.println("after delete NO="+num);
        o=null
        System.gc();                    //手工方式回收无用的对象
     }
     public static void main(String [] args) {
         Counter [] objs=new Counter[5];
         for(Counter obj: objs) {
             obj=new Counter();
         }
         Counter.delete(objs[1]);
         Counter.delete(objs[0]);
     }
}
```

程序运行结果：

```
after new NO=1
after new NO=2
after new NO=3
after new NO=4
after new NO=5
after delete NO=4
after delete NO=3
```

4.5 包

包的出现，主要是因为类的种类非常多，类的命名很有可能重复，否则只能让类的命名越来越长，越来越复杂，Java 使用包的机制很好地解决了这个问题。在一个包中，类是不可以重名的，但是不同的包中允许相同的类名出现，通过使用“包名＋类名”的方式，可以达到有效管理类的目的。

4.5.1 包的定义

创建一个包非常简单，只需在源文件的起始处添加一条包含关键字 package 的语句即

可,则任何在该文件中定义的类都属于指定的包。package 语句指定了一个类存放的命名空间,如果没有 package 关键字,则类会被放入一个默认的包中,该包没有名字。

包的定义格式为:

```
package  [包名 1.[包名 2.[包名 3]]]
```

包定义语句中,用.指明包的层次,Java 编译器根据上述定义将包一一映射为文件系统的目录结构。例如,声明 package First.Second.Third 的包,则该类文件将存储在文件系统的当前 Classpath 下的 First\Second\Third 子目录中。

假定当前名为 Hello 的 project 的根目录为 H:\test,即当前系统路径为 H:\test,在该 project 中生成包 package First.Second.Third 和文件 MainTest.java 后,其对应的目录结构如图 4-6 所示。



图 4-6　包的层次和文件系统目录间的关系

4.5.2　包的引入

如果包已经存在,可以通过关键字 import 引入包中的任何类,并使得这些类在整个程序中可见。import 语句的基本格式为:

```
import [包名 1.[包名 2.[包名 3.]]] (类名|*)
```

与 package 相似,[包名 1.[包名 2.[包名 3.]]]为包的层次,实质为文件系统的目录结构,(类名|*)表示只引入指定类名的类,或者引入该包中所有的类(*)。如果要使用 Java 类库中的已有类或其他包中的类,必须先通过 import 语句将其引入才能使用。例如:

```
import java.util.*;                    //引入 java.util 包中的所有类
import java.applet.Applet;             //引入 java.applet 包中的类 Applet
Date day1=new Date();
```

【例 4.12】　计算任意两个点之间的距离。

```
package code0405.mypack;
//定义包 code0405.mypack 中的类 NewPoint
public class NewPoint {
    public double x, y;

    public NewPoint(double a, double b) {
        x=a;
        y=b;
    }
```

```
        public double distanceTo(NewPoint p) {        //该点到另一点 p的距离
            return Math.sqrt((x-p.x) * (x-p.x)+(y-p.y) * (y-p.y));
        }
}
package code0405;
import code0405.mypack.NewPoint;    //引入包 code0405.mypack中的类 NewPoint
public class UsePoint {
    public static void main(String[] args) {
        NewPoint p1=new NewPoint(1.0, 2.0);        //创建两个 NewPoint 对象
        NewPoint p2=new NewPoint(2.0, 5.5);
        System.out.println("点 p1 坐标: "+p1.x+", "+p1.y);
        System.out.println("点 p2 坐标: "+p2.x+", "+p2.y);
        System.out.println("点 p1 到点 p2 的距离: "+p1.distanceTo(p2));
    }
}
```

程序运行结果：

```
点 p1 坐标: 1.0, 2.0
点 p2 坐标: 2.0, 5.5
点 p1 到点 p2 的距离: 3.640054944640259
```

分析：本例定义了两个 Java 类：类 NewPoint 用于描述点信息，放在包 code0405. mypack 中；类 UsePoint 作为主类，放在包 code0405 中。由于两个类不在同一包中，因此必须先通过 import 语句将 NewPoint 引入到 UsePoint 中，否则它对 UsePoint 是不可见的。

如果使用的两个类都在同一个包中（如将 UsePoint 中起始语句改为 package code0405. mypack)，则无须进行包的引入，即可在 UsePoint 中访问 NewPoint。

提示：如果不使用 import 语句，则必须在每个被引用类前加上包名作为前缀。例如：code0405. mypack. NewPoint p1＝new code0405. mypack. NewPoint (1. 0, 2. 0);，虽然这与使用 import 的方法是等价的，但相比而言，import 语句使程序更简洁。

4.6 类及成员修饰符

如果对类及其成员访问有特殊限制，可以引入修饰符对类及其成员进行限定，以说明它们的性质、相互关系和适用范围。常见的修饰符包括 public、protected、private、final、abstract 和 static 等，本节将重点介绍前 4 种修饰符，abstract 将在下一章进行介绍。

4.6.1 public

public 可以同时修饰类和成员变量、成员方法。如果 public 修饰一个类名，如 public class A{…}，则表示该类可以被所有的其他类访问或引用，即其他类中可以创建该类的实例，访问该类的所有可见成员变量，调用所有可见方法。

如果类成员(变量和方法)用 public 修饰，表示该类的成员不仅可以被其内部成员访问，而且可以被其他类直接访问，也就是说，外界可直接存取公有数据和公有方法。

【例 4.13】 public 使用示例。

```
package code0406;

class Pub {
    public int x;                        //公有数据
    public void set_x(int i) {           //公有方法
        x= i;
    }
    public void show_x() {               //公有方法
        System.out.println("x= "+ x);
    }
}

//类 PubTest
package code0406;

class PubTest {
    public static void main(String args[]) {
        Pub obj= new Pub();
        obj.set_x(3);
        obj.show_x();
        obj.x= 10;                //在 PubTest 中访问类 Pub 中的公有数据
        System.out.println("x= "+ obj.x);
    }
}
```

程序运行结果:

```
x= 3
x= 10
```

分析:类 Pub 中所有的成员都是 public 的,因此其中的成员变量 x 和成员方法 set_x()和 show_x()都可直接被另一类 PubTest 访问。

4.6.2 protected

关键字 protected 主要用于修饰类成员,说明该成员是被保护成员,除了可以被类自身访问外,还可以被该类的子类及与该类在同一个包中的其他类访问。

【例 4.14】 protected 使用示例。

```
package code0406;

class Prot {
    protected int n= 3;                  //被保护变量
    protected void show_n() {            //被保护方法
    System.out.println("n= "+ n);
    }
}

package code0406;
```

```
class ProtTest extends Prot {                 //生成子类
    public static void main(String args[]) {
        ProtTest abs=new ProtTest();
        abs.show_n();                         //访问父类的被保护成员
        abs.n=10;                             //访问父类的被保护成员
        abs.show_n();
    }
}
```

程序运行结果：

```
n=3
n=10
```

分析：该例演示的是在子类中访问父类中受保护的成员。ProtTest 是 Prot 的子类，关键字 extends 表示它们之间是继承关系（继承参见第 5 章）。可以看出，子类可以直接访问父类中的被保护数据和方法。

4.6.3 private

关键字 private 主要用于修饰类成员，表示该类成员只能被类自身访问，任何其他类（包括该类的子类）都无权修改或引用。应该将不希望他人随意引用或修改的数据和方法设置为 private，这将使得私有成员对其他类不可见，只有通过声明为 public 的方法才可以对这些数据进行访问，从而达到信息隐藏和封装的目的。

【例 4.15】 private 使用示例。

```
package code0406;
public class MyDate {
    private int day, month, year;                 //定义私有变量
    public void tomorrow() {
        day=day+1;                                //本类可以引用私有变量
    }
    public void setDay(int d) {
        day=d;
    }
}
package code0406;
public class MyDateUser {
    public static void main(String args[]) {
        MyDate mydate=new MyDate ();
        mydate.day=21;                            //错误
        System.out.println("day="+mydate.day);    //错误
        mydate.setDay(21);                        //正确
    }
}
```

分析：类 MyDate 中 3 个成员变量都被 private 修饰，在本类中对私有成员的使用不受

任何限制,但在其他类,如 MyDateUser 中,对 MyDate 的私有成员使用是受限的。因此,语句 mydate.day=21 是错误的。如果想对其中某个私有成员变量进行操作,可以借助类 MyDate 的公有方法完成,如 mydate.setDay(21)也可达到将 day 赋值为 21 的目的。

提示:如果类成员前没有 public、protected、private 中的任何一个修饰符,则称它使用了默认(default)修饰符。这时,只有该类本身以及与该类在同一个包中的其他类才可以直接访问这些默认成员。

表 4-1 对上述修饰符的使用进行了总结。

表 4-1　Java 类的成员变量和方法访问权限

	同一个类	同一个包	不同包中的子类	不同包中的非子类
private	★			
default	★	★		
protected	★	★	★	
public	★	★	★	★

4.6.4　final

关键字 final 可用于修饰类、成员变量、成员方法。final 的基本意思是"最终的",即它所修饰的元素不允许再修改。例如,用 final 修饰的类(叫最终类)不能再有子类;用 final 声明的方法(最终方法)不能再被重写;用 final 声明的成员变量(常量)初始化后,不能再被重新赋值或修改。

【例 4.16】　final 使用示例。

```
package code0406;

final class Circle {
    final double PI=3.1416;
    final double area(double r) {
        return (PI * r * r);
    }
}
package code0406;
class FinalTest {
    static public void main(String arg[]) {
        Circle c=new Circle();
        c.PI=2.0;                   //错误
        System.out.println("area="+c.area(5.0));
    }
}
```

分析:类 Circle 中所有成员变量和方法均被 final 修饰,并且类 A 本身也是 final 的。因此,如果生成类 A 的子类或在子类中重写方法 area()是不允许的,类似地,语句 a.PI=2.0 对常量进行修改也是错误的。

4.6.5 实用案例

【例 4.17】 实现一简单的员工信息簿。

可以向该信息簿中添加员工基本信息，如姓名、工资等，同时显示该信息簿中所有员工的基本信息。

本实用案例程序如下：

```
package code0406;

public class Employee {
    private String name;                    //员工的姓名
    private double salary;                  //员工的工资
    Employee(){                             //构造方法 1
        name="unknown";
        salary=0;
    }
    Employee(String n, double s){           //构造方法 2
        name=n;
        salary=s;
    }
    public String getName() {               //获取员工的姓名
        return name;
    }
    public void setName(String name) {      //设置员工的姓名
        this.name=name;
    }
    public double getSalary() {             //获取员工的工资
        return salary;
    }
    public void setSalary(double salary) {  //设置员工的工资
        this.salary=salary;
    }
}

package code0406;

public class EmployeeDirectory {
    public static void main(String[] args) {
        Employee [] dir=new Employee[4];
        Employee e1=new Employee();         //通过构造方法 1先生成对象 e1
        e1.setName("ZhangShan");            //再通过 set 方法为 e1 赋值
        e1.setSalary(100);
        dir[0]=e1;
        dir[1]=new Employee();              //使用默认值和构造方法 1对对象初始化
        for(int i=2; i<=3;i++){
            dir[i]=new Employee("user"+i,200);      //通过构造方法 2完成对象初始化
        }
```

```
            for(int j=0; j<=3;j++){                         //输出员工消息
                System.out.println("员工的姓名:"+dir[j].getName());
                System.out.println("员工的工资:"+dir[j].getSalary());
            }

        }
    }
```

程序运行结果:

```
员工的姓名:ZhangShan
员工的工资:100.0
员工的姓名:unknown
员工的工资:0.0
员工的姓名:user2
员工的工资:200.0
员工的姓名:user3
员工的工资:200.0
```

4.7 类和对象实训任务

【任务描述】

模拟银行 ATM 机,编写一具有简单操作界面的 Java 应用程序,实现包括存款、取款、查询等功能在内的简单应用。

【任务分析】

(1) 想象现实世界中的相关操作涉及哪些实体呢?一个是储户,一个是 ATM 机。因此根据面向对象中的抽象原则,可将其抽象为两个类:代表储户的账户信息类,代表银行 ATM 机的 ATM 类。外加一个主类(负责实例化其他类的对象)。

(2) Account 类记录储户的卡号、姓名、密码和账户余额等信息,并提供 get 方法获取每个属性的值,对账户余额属性提供 sub_Balance()方法和 add_Balance()方法以模拟余额增加、减少的功能。

注意:为加强对属性数据的控制访问,考虑将其定义为 private。

(3) ATM 类模拟 ATM 机的主要功能,根据对银行 ATM 机的了解,考虑设计以下主要方法。

① Welcome()方法:欢迎显示功能。

② Load_Sys()方法:登录功能。

③ SysOpter()方法:根据用户输入进行任务调度。

④ Inqu_Info()方法:查询用户账户。

⑤ BetBalance()方法:取款。

⑥ AddBalance():存款。

⑦ isBalance()方法:判断余额是否足够。

⑧ isRight()方法:判断卡号、密码是否正确。

(4) Test 类为主类，是整个应用执行的入口，主要完成创建 ATM 类实例的功能，并通过对 Welcome()和 Load_Sys()方法的调用显示欢迎界面，提示用户完成系统登录操作。

【任务解决】

完整的示例程序如下：

```
/* Account 类封装储户信息及部分功能 * /
package code0407;
import java.io.* ;
class Account {
    private String number=null;                  //卡号
    private String name=null;                    //客户姓名
    private String password=null;                //客户密码
    private double money=0.0;                    //余额
    /***** 构造方法,以生成多个储户信息 ***********/
    public Account(String number, String name, String password, double money) {
        this.number=number;
        this.name=name;
        this.password=password;
        this.money=money;
    }
    protected String get_number() {
        return number;
    }
    protected String get_Name() {
        return name;
    }
    protected String get_Password() {
        return password;
    }

    public double get_Money() {
        return money;
    }
    protected void sub_Balance(double mon) {     //余额减少
        money -=mon;
    }
    protected void add_Balance(double mon) {     //余额增加
        money+=mon;
    }
}

/* ATM 类模拟 ATM 机的主要功能 * /
package code0407;
import java.io.* ;
```

```
class ATM {
    Account act;
    public ATM() {
        act=new Account("000", "test", "111", 5000); //生成 Account 实例
    }
    /*********** 欢迎界面 ***********/
    protected void Welcome() {
        String str="---------------------------------";
        System.out.print(str+"\n");
        System.out.print(" 1.取款."+"\n"+" 2.查询."+"\n"+" 3.存款."+"\n"+" 4.退出系统."+"\n");
        System.out.print(str+"\n");
    }
    /********** 登录系统 **********/
    protected void Load_Sys() throws Exception {
        String card, pwd;
        int counter=0;
        BufferedReader br=new BufferedReader(new InputStreamReader(System.
        in));                              //创建标准输入输出流,详见第 8 章
        do {
            System.out.println("请输入您的卡号:");
            card=br.readLine();            //读取键盘输入信息
            System.out.println("请输入您的密码:");
            pwd=br.readLine();
            if(!isRight(card, pwd)) {
                System.out.println("您的卡号或密码输入有误.");
                counter++;
            } else
                SysOpter();

        } while(counter<3);
        System.exit(1);                    //应用退出
    }
    /********** 系统操作提示 **********/
    protected void SysOpter() throws Exception {
        int num;
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
        System.out.println("请选择您要操作的项目(1-4):");
        num=br.read();                     //num 为 ASCII 码转换的整数
        switch (num) {
        case 49:
            BetBalance();
            break;
        case 50:
            Inqu_Info();
```

```
            break;
        case 51:
            AddBalance();
            break;
        case 52:
            Exit_Sys();
            break;
        }
        System.exit(1);
}

/********** 信息查询 **********/
protected void Inqu_Info() throws Exception {
    System.out.print("---------------------\n"+"账号:"+act.get_number()+"\n"+"姓
    名:"+act.get_Name()+"\n"
        +"余额:"+act.get_Money()+"\n"+"-----------------------\n");
    SysOpter();
}
/********** 取款 **********/
public void BetBalance() throws Exception {
    String str=null, str1;
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    do {
        System.out.println("请输入取款数目:");
        str=br.readLine();
        double qu=Double.valueOf(str).doubleValue();          //将字符串转换为 double 类型
        if (qu>act.get_Money()) {
            System.out.println("余额不足,请重新输入您要取的数目:");
        } else {
            act.sub_Balance(qu);
            System.out.println("取款成功,您的账户余额为:"+act.get_Money());
            Welcome();
            SysOpter();
        }
    } while(true);
}

/********** 存款 **********/
public void AddBalance() throws Exception {
    String str=null, str1;
    BufferedReader br=new BufferedReader(new InputStreamReader(System.in));
    do {
        System.out.println("请输入存款数目:");
```

```
                str=br.readLine();
                double qu=Double.valueOf(str).doubleValue();
                act.add_Balance(qu);
                System.out.println("存款成功,您的账户余额为:"+act.get_Money());
                Welcome();
                SysOpter();
            } while(true);

        }
        /*********** 判断卡内是否有钱 ***********/
        protected boolean isBalance() {
            if (act.get_Money()<0) {
                return false;
            }
            return true;
        }
        /********* 卡号密码是否正确 ******/
        protected boolean isRight(String card, String pwd) {
            if (act.get_number().equals(card) && act.get_Password().equals(pwd))
                return true;
            else
                return false;
        }
        /*********** 结束系统 ***********/
        protected void Exit_Sys() {
            System.out.println("感谢您使用本系统,再见!");
            System.exit(1);
        }
    }

    /* ATMTest 类为主类 */
    package code0407;
    public class ATMTest {
        public static void main(String[] args) throws Exception {
            ATM atm=new ATM();
            atm.Welcome();
            atm.Load_Sys();
        }
    }
```

习题与思考

1. 在Java类中定义两个或更多方法，它们有相同的方法名而参数不同时，这称为(　　)。

A. 继承　　B. 多态性　　C. 构造方法　　D. 方法重载

2. 定义一个时钟类(Clock)，要求如下：

(1) 存储时钟的时 hour(0～23)、分 minute(0～59)、秒 second(0～59)。

(2) 创建新对象时默认为0时0分0秒。

(3) 设置时钟为指定的时间。

(4) 使时钟前进1秒钟的方法 incSecond()。

(5) 以“时:分:秒”的形式输出时钟的时间。

3. 编写一个 Java Application 程序，使用复数类 Complex 验证两个复数 1+2i 和 3+4i 相加产生1个新的复数 4+6i。复数类 Complex 必须满足如下要求：

(1) 复数类 Complex 的属性有：

RealPart：int 型，代表复数的实数部分。

ImaginPart：int 型，代表复数的虚数部分。

(2) 复数类 Complex 的方法有：

Complex()：构造函数，将复数的实部和虚部都置0。

Complex(int r,int i)：构造函数，形参 r 为实部的初值，i 为虚部的初值。

Complex complexAdd(Complex a)：将当前复数对象与形参复数对象相加，所得的结果仍是一个复数值，返回给此方法的调用者。

String ToString()：把当前复数对象的实部、虚部组合成 a+bi 的字符串形式，其中 a 和 b 分别为实部和虚部的数据。

4. 给出下列程序运行的结果：

```
public class OverloadDemo {
    void testOverload(int i) {
        System.out.println("int");
    }
    void testOverload(String s) {
        System.out.println("String");
    }
    public static void main(String args[]) {
        OverloadDemo a=new OverloadDemo();
        char ch='x';
        a.testOverload(ch);
    }
}
```

5. Java 中类成员的访问修饰符有哪些？它们各有什么作用？

第 5 章 继承与多态

本章学习目标

继承是一种基于已有类创建新类的机制，利用继承可以先创建一个具有广泛意义的类，然后通过派生创建新类，并添加一些特殊的属性和行为。被继承的类称为父类，其派生得到的类称为子类。类的继承是实现代码复用最有效的方法，本章将主要讲述类继承过程中的几个关键问题：

(1) 继承的实现。
(2) 方法的重写。
(3) 多态的表现。
(4) 抽象类与抽象方法。
(5) 接口。
(6) 泛型。
(7) 内部类。
(8) Java 反射机制。

5.1 继承使用初探

当新建一个类时，也许会发现该类与之前的某个类非常相似，如绝大多数的属性和行为都相同。这时，可以选择复制原类中的语句，对其部分修改后加入新类中，但这意味着必须同时维护两个相似的 Java 程序。

另外一种方法是通过继承，让新类自动获得被继承类中已有的属性和方法，同时添加原类中没有的属性和方法即可。例如，根据人的特征定义类 Person 如下：

```
public class Person {
    private String name;
    private int age;
    public void Say(){
        System.out.println(name+ "can say");
    }
```

```
    public void setName(String name) {
        this.name=name;
    }
    public String getName(){
        return name;
    }
}
```

该类对所有人均适用,但如果根据学生的特点需要定义一学生类,则可以肯定的是学生类中,除了姓名、年龄属性外,还可能有所在学校名称(这是一般人没有的),此外,学生的行为中还包括上学(GotoSchool)这一方法,这也是一般人没有的。因此,可以通过继承的方式建立类 Student 如下:

```
public class Student extends Person{
    String schoolname;                          //增加新的属性 schoolname
    public void GotoSchool(){                   //增加新的方法 GotoSchool
        System.out.println("I am going to school");
    }
    public static void main(String[] args) {
        Student student1=new Student();
        student1.name="MingM";student1.age=10;  //引用继承自父类的变量
        student1.schoolname="CQ";
        student1.Say();                         //调用继承自父类的方法
        student1.GotoSchool();                  //调用子类新增加的方法
        System.out.println("My name is "+student1.name);
        System.out.println("My schoolname is "+student1.schoolname);
    }
}
```

程序运行结果:

```
I can say
I am going to school
My name is MingM
My schoolname is CQ
```

分析:通过关键字 extends 定义了类 Person 的子类 Student,然后添加了只有学生才有的属性 schoolname 和方法 GotoSchool()。

在 main()方法中,尽管 Student 中没有定义变量 name、age 以及方法 Say(),但是子类却可以通过继承的方式自动取得,并像访问自己的成员变量和方法一样引用即可。

5.2 类的继承

5.2.1 继承的实现

继承的实现其实非常简单,其语法格式如下:

```
class 子类名 extends 父类名{
    类体
}
```


extends 是关键字,后跟父类的类名,如果没有父类,则默认父类是 java. lang. Object。Java 只支持单继承,即只能有一个父类,但类之间的继承可以具有传递性。

子类可以通过继承自动获得父类中访问权限为 public、protected、default 的成员变量和方法,但不能继承权限为 private 的成员变量和方法。

【例 5.1】 类的继承示例。

```
package code0502;

class A {
    int i;
    void showi() {
        System.out.println("i: "+i);
    }
}
class B extends A {
    int k;

    void show() {
        System.out.println("k: "+k);
        showi();
    }
    void sum() {
        System.out.println("i+k: "+(i+k));
    }
}
public class Simple {
    public static void main(String args[]) {
        A superOb=new A();
        B subOb=new B();
        superOb.i=10;              //对父类对象的成员赋值
        System.out.println("Contents in 父类: ");
        superOb.showi();
        subOb.i=7;                 //对子类中继承得到的变量 i 赋值
        subOb.k=9;
        System.out.println("Contents in 子类: ");
        subOb.show();
        System.out.println("Sum of i and k in 子类:");
        subOb.sum();
    }
}
```

程序运行结果:

```
Contents in 父类:
i: 10
Contents in 子类:
k: 9
i: 7
Sum of i and k in 子类:
i+k: 16
```

分析：类B中虽然没有定义变量i和方法showi()，但却可以通过继承关系获得，因此在类B中直接引用i和方法showi()是正确的。但是，如果在类A的语句int i前加上修饰符private，则上述程序会出现编译错误，其原因在于类B无法继承到private类型的变量i，因此i对类B是不可见的。

提示：尽管一个子类可以从父类继承所有允许的方法和变量，但它不能继承构造函数，掌握这一点很重要。一个类要得到构造函数，只有两个办法：一是重写构造函数；二是根本不写构造函数，这时系统为每个类生成一个默认构造函数。

5.2.2 继承与重写

在类的继承过程中，如果子类中新增的变量和方法与父类中原有的数据和方法同名，则会重写(也称覆盖)从父类继承来的同名变量和方法。重写又分变量重写和方法重写，变量重写是指父类和子类中的变量名相同，数据类型也相同。方法重写与之前介绍的方法重载相似，但更严格：不仅要求父类与子类中的方法名称相同，而且参数列表也要相同，只是实现的功能不同。

【例5.2】 重写父类中的同名方法和变量。

```
package code0502;

class SuperCla {
    int a=3, b=4;
    void show() {
        System.out.println("super result="+ (a+b));
    }
}
class SubCla extends SuperCla {
    int a=10;                    //重写了父类中同名的变量 a
    void show() {                //重写了父类中同名的方法 show
        int c=a * b;
        System.out.println("sub result="+c);
    }
}
public class OverrideTest {
    public static void main(String args[]) {
        SuperCla sp=new SuperCla();
        SubCla sb=new SubCla();
        sp.show();               //此处调用的是父类中的方法 show
        System.out.println("In super Class:a="+sp.a);
```

```
                                    //此处引用的是父类中的变量 a
        sb.show();                  //此时子类对象的 show 方法覆盖了父类的同名方法
        System.out.println("In sub Class:a= "+ sb.a);
    }
}
```

程序运行结果:

```
super result= 7
In super Class:a= 3
sub result= 40
In sub Class:a= 10
```

分析:子类 SubCla 中定义有与父类 SuperCla 同名的变量 a 和方法 show(),因此,使用子类对象 sb 访问变量 a 和方法 show()时,引用的是子类中的成员,父类的同名变量被覆盖,同名的方法被重写。

如果想在子类中访问父类中被覆盖的成员,可以使用关键字 super 来解决这一问题,其基本格式如下:

```
访问父类成员 :super.成员变量
```

或

```
super.成员方法 ([参数列表])
```

访问父类构造方法:

```
super([参数列表])
```

【例 5.3】 super 关键字的使用。

```
package code0502;
//定义员工类
class Employee {
    private String name;
    private int salary;
    public String getDetails() {
        return "Name:"+ name+ "\nSalary:"+ salary;
    }
    Employee() {
        name= "Tom";
        salary= 1234;
    }
}
//定义经理类
class Manager extends Employee {
    public String department;
    /* 重写 getDetails 方法 */
    public String getDetails() {
```

```
            System.out.println("I am in Manager");
            return super.getDetails();      //调用父类的 getDetails 方法
        }
        Manager() {
            super();                        //访问父类的无参构造方法,即 Employee()
            department="sale";
        }
}
public class Inheritance {
        public static void main(String arg[]) {
            Manager m=new Manager();
            System.out.println(m.getDetails());
            System.out.println("department:"+m.department);
        }
}
```

程序运行结果：

```
I am in Manager
Name:Tom
Salary:1234
department:sale
```

分析：程序首先对 Manager 实例化，并自动调用无参构造方法 Manager()，主要完成两项主要任务：一是通过 super()调用父类的无参构造函数，即 Employee()；二是对子类中新增变量 department 初始化。接着，语句 m. getDetails()调用子类的同名方法 getDetails()，并在方法体中通过 super. getDetails()实现对父类中 getDetails()方法的调用。

注意：生成一个子类的实例时，首先要执行子类的构造方法，但如果该子类继承自某个父类，则执行子类的构造方法前，系统将先自动调用父类的无参构造函数。因此，在例 5.3 中的方法 Manager()中去掉语句 super()效果是完全相同的。对于父类含参数的构造方法，子类可以通过在自己的构造方法中使用 super 关键字来调用它，但这个调用语句必须是子类构造方法中的第一条可执行语句。

与 super 不同，关键字 this 的主要作用在于表示当前对象的引用，当局部变量和类的成员变量同名时，该局部变量作用区域内成员变量就被隐藏了，必须使用 this 来指明。

【例 5.4】 this 关键字的使用。

```
package code0502;

public class ThisTest {
        public static void main(String[] args) {
            Local aa=new Local();
        }
}
class Local {
        public int i=1;                     //这个 i 是成员变量
        Local(int i) {                      //这个 i 是局部变量
```

```
            System.out.println("this.i= "+ this.i);            //this.i 指的是对象本身的成员变量 i
            System.out.println("i= "+ i);          //变量 i 前没有 this,因此是局部变量
        }
        Local(){
            this(6);
        }
}
```

程序运行结果:

```
this.i= 1
i= 6
```

分析:关键字 this 的主要作用有两点:一是通过它引用成员变量,如上例中 this.i 指的是当前对象中的成员变量 i;二是通过 this 调用类的构造方法,如上例中的 this(6)将调用对应的构造方法 Local(int i)。

5.2.3 继承与类型转换

和标准类型数据的转换一样,不同类的对象之间也可以相互转换,但前提是源和目标类之间必须通过继承相联系。转换可分为显式和隐式转换两种。显式转换格式为:

```
(类名)对象名
```

它将对象转换成类名所表示的其他对象。Java 支持父类和子类对象之间的类型转换,如果是子类对象转换为父类,可进行显式转换或隐式转换;如果是父类对象转换成子类,编译器首先要检查这种转换的可行性,如果可行,则必须进行显式转换。

【例 5.5】 对象类型的转换示例。

```
package code0502;
class CA{
    String s= "class CA";
}
class CB extends CA{
    String s= "class CB";
}
class Convert{
    public static void main(String args[]){
        CB bb,b= new CB();
        CA a,aa;
        a= (CA)b;                    //显式转换
        aa= b;                       //隐式转换
        System.out.println(a.s);
        System.out.println(aa.s);
        bb= (CB)a;                   //显式转换
        System.out.println(bb.s);
    }
}
```

程序运行结果：

```
class CA
class CA
class CB
```

分析：b是子类CB的实例，将其转换为父类CA的实例时可以进行显式或隐式转换。而父类对象a转换为子类CB的对象时，必须进行显式转换。

5.2.4 实用案例

【例5.6】 通过继承定义员工和经理类。

普通员工和经理作为职员有很多共同之处，如都可以取得工资报酬，但经理可能还会获得额外的奖金，因此在成员属性和方法上可能有特殊之处。可以将NewEmployee类定义为父类，NewManager类作为子类，子类继承了父类，并添加了一个新的setBonus方法，用于增加奖金，最后打印出结果。

本实用案例程序如下：

```
package code0502;
import java.util.* ;
class NewEmployee {
    private String name;
    private double salary;
    private Date hireDay;

    public NewEmployee(String n, double s, int year, int month, int day) {
        name= n;
        salary= s;
        GregorianCalendar calendar= new GregorianCalendar(year, month- 1, day);
        hireDay= calendar.getTime();
    }
    public String getName() {
        return name;
    }
    public double getSalary() {
        return salary;
    }
    public Date getHireDay() {
        return hireDay;
    }
    public void raiseSalary(double byPercent) {
        double raise= salary * byPercent/100;
        salary+ = raise;
    }
}
```

```
package code0502;

class NewManager extends NewEmployee {
    private double bonus;
    public NewManager(String n, double s, int year, int month, int day) {
        super(n, s, year, month, day);
        bonus=0;
    }
    public double getSalary() {
        double baseSalary=super.getSalary();
        return baseSalary+bonus;
    }
    public void setBonus(double b) {
        bonus=b;
    }
}

package code0502;

public class ManagerTest {
    public static void main(String[] args) {
        NewEmployee e=new NewEmployee("Harry Hacker", 50000, 1989, 10, 1);
        e.getName();
        System.out.println(e.getName()+":"+e.getSalary());
        NewManager boss=new NewManager("Carl Cracker", 80000, 1987, 12, 15);
        boss.setBonus(5000);
        System.out.println(boss.getName()+":"+boss.getSalary());

    }
}
```

程序运行结果:

```
Harry Hacker:50000.0
Carl Cracker:85000.0
```

5.3 多 态 性

5.3.1 多态性的概念

简单地说,多态性就是一个名称可以对应多种不同的实现方法。Java 的多态性体现在两个方面:编译多态和运行多态。

编译多态是指在程序编译过程中体现出的多态性,如方法重载,尽管方法名相同,但由于参数不同,在调用时系统根据传递参数的不同确定被调用的方法,这个过程是在编译是完成的。例如,定义一个作图的类,它有一个 draw()方法用于绘图,根据不同的使用情况,可以接收字符串、矩形、圆形等参数。对于每一种实现,方法名都为 draw(),只不过具体实现

方式不同，不用另外重新起名，这样大大简化了方法的实现和调用，程序员无须记住很多的方法名，只需传递相应的参数即可。

运行多态则是由类的继承和方法重写引起的，由于子类继承了父类的属性和方法，因此，凡是父类对象可以使用的地方，子类对象也可以使用。如例5.5中的类CA和CB，可以直接生成子类CB的对象，并将该引用赋给父类CA的对象，即：

```
CA a=new CB();
```

它等价于下面两个子句：

```
CB b=new CB(); CA a=b;          //隐式类型转换
```

现在有一个问题：如果子类重写了父类的成员方法，调用该方法时，到底应该调用父类中的方法还是子类中的方法？这无法在编译时确定，需要系统在运行时根据实际来决定，所以这种由方法重写引起的多态称为运行多态。

Java规定：对重写的方法，Java根据调用该方法的实例的类型来决定选择哪个方法。对子类的实例，如果子类重写了父类的方法，则调用子类的方法；如果子类没有重写父类的方法，则调用父类的方法。

【例5.7】 类的多态性示例。

```
package code0503;
class A {
    void callme() {
        System.out.println("inside A");
    }
}
class B extends A {
    void callme() {
        System.out.println("inside B");
    }
}
class Poly {
    public static void main(String args[]) {
        A a=new A();
        B b=new B();
        A c=new B();
        a.callme();
        b.callme();
        c.callme();
    }
}
```

程序运行结果：

```
inside A
inside B
```

```
inside B
```

分析：对实例 a 和 b，它们的类型为类 A 和 B，所以分别调用父类和子类中的方法 callme()即可。对实例 c，它被实例化为子类 B 的一个对象，且子类重写了方法 callme()，因此根据转型规则，(由于调用该方法的实例类型为 B)这时应该调用子类 B 的方法，所以输出 inside B。

多态性在实际应用中有好处很多，例如：

(1) 可替换性。多态对已存在代码具有可替换性。例如，多态对圆 Circle 类有效，对其他任何圆形几何体(如圆环)也同样有效。

(2) 可扩充性。多态对代码具有可扩充性。增加新的子类不影响已有类的多态性、继承性。实际上，新加子类更容易获得多态功能。例如，在实现了圆锥、半圆锥以及半球体的多态基础上，很容易增添球体类的多态性。

5.3.2 实用案例

【例 5.8】 将员工和经理存入同一数组。

数组一般要求存入的数据必须是同一类型，如整型数组要求每个数组成员均为整数。但之前定义的类 NewEmployee 和 NewManager 显然不是同一类型，那如何将其存入到同一个数组中呢？

本实用案例程序如下：

```
package code0502;

public class EmployeeArray {
    public static void main(String[] args) {
        NewEmployee[] staff=new NewEmployee[3];
        NewManager boss=new NewManager("Carl", 80000, 1987, 12, 15);
        boss.setBonus(5000);
        staff[0]=boss;
        staff[1]=new NewEmployee("Harry", 50000, 1989, 10, 1);
        staff[2]=new NewEmployee("Tommy", 40000, 1990, 3, 15);
        for (NewEmployee e: staff)
            System.out.println("name="+e.getName()+",salary="+e.getSalary());
    }
}
```

程序运行结果：

```
name=Carl,salary=85000.0
name=Harry,salary=50000.0
name=Tommy,salary=40000.0
```

分析：满足上述需求的关键在于数组类型的定义，如本例中将数组声明为父类 NewEmployee 类型，这样将子类对象存入数组时，系统将利用隐式类型转换规则，将其统一到父类类型中，即可实现将不同类对象存入同一数组的目标。

5.4 抽象类与抽象方法

5.4.1 定义抽象类及实现抽象方法

关键字 abstract 修饰的类称为抽象类，抽象类是一种没有完全实现的类。不能用它实例化任何对象，它的主要用途是用来描述一些概念性的内容，然后在子类中具体去实现这些概念，这样可以提高开发效率，统一用户接口，所以抽象类更多是作为其他类的父类。

抽象类中可以含有抽象方法，抽象方法是用 abstract 修饰的方法，抽象方法只有方法的返回值、名称和参数列表，没有方法体，它必须在子类中具体实现该方法(即给出方法体)。

【例 5.9】 定义抽象类和它的具体实现。

```
package code0504;

abstract class Abs {
    abstract void show();
    abstract void show(int i);
}
/* 定义抽象类 Abs 的子类 Real */
public class Real extends Abs {
    int x;
    void show() {          //实现抽象方法 abstract void show()
        System.out.println("x= "+x);
    }
    void show(int i) {     //实现抽象方法 abstract void show(int i)
        x=i;
        System.out.println("x= "+x);
    }
}
```

分析：Abs 是一个抽象类，其中两个方法也是抽象的，没有任何代码。子类 Real 中重写了这两个方法，并给出具体的实现。

注意：如果一个类包含抽象方法，则必须被定义为抽象类，但抽象类不一定要包含抽象方法。

5.4.2 实用案例

【例 5.10】 通过抽象类实现自定义堆栈。

本实用案例程序如下：

```
package code0504;

abstract class Access{          //定义一个抽象的类 Access,提供基本的存入、取出操作
    abstract void put(char c);
    abstract char get();
}
```

```
package code0504;
class LinkedList {                          //定义双向链表类
    char data;
    LinkedList back;
    LinkedList forward;
}
package code0504;
class MyStack extends Access {              //通过继承定义堆栈类
    private LinkedList bottom=new LinkedList();
    private LinkedList top=bottom;          //初始化栈顶与栈底

    /**实现接口 Access 的 put()方法,该方法向栈存一个字符 */
    public void put(char c){
        top.forward=new LinkedList();
        top.forward.data=c;
        top.forward.back=top;
        top=top.forward;
    }

    /**实现接口 Access 的 get()方法,该方法从栈中取一个字符 */
    public char get(){
        if(top!=bottom){
          char ch=top.data;
          top.back.forward=null;
          top=top.back;
          return ch;
        }
        else{
          System.out.println("The stack is empty!");
          return '\0';
        }
    }
}
package code0504;
public class DataTest {
    public static void main(String args[]){
        MyStack s=new MyStack();
        s.put('x');
        s.put('y');
        s.put('z');                   //向栈 s 中存入 3个字符
        System.out.println("In Stack:");
        System.out.println(s.get());
        System.out.println(s.get());
        System.out.println(s.get());        //从栈 s 中取数并显示
```

```
    }
}
```

程序运行结果：

```
In Stack:
z
y
x
```

分析：基于抽象类 Access 的定义，可以通过继承产生更多相似的子类，如队列、树结构等。每个子类虽然在完成数据存入、取出时的具体逻辑不同，但均保持相同的访问接口形式，如 get()、put()。

5.5 接　口

5.5.1 接口定义

接口是抽象类的一种变体，但它与抽象类又有一些差异。首先，接口是用关键字 interface 来定义的，其形式为：

```
[修饰符] interface 接口名
{
…//变量和方法声明
}
```

其次，接口中的所有方法都是抽象的(abstract 可以省略)，也没有方法体。看起来与抽象非常相似，但它们仍然有显著的区别：

(1) 抽象类中的方法不一定都是抽象的，而接口中的所有方法都是抽象的。

(2) 接口中的所有数据都是公有静态常量：即具有 public、static、final 属性，但一般不写这些修饰符。

(3) 因为接口中所有方法都是抽象的、public 的，因此必须在实现该接口的子类中重写所有这些方法。

例如：

```
interface Myinter
{
    double g=9.8;          //等价于  public static final double g=9.8
    void show();           //等价于  public abstract void show()
}
```

5.5.2 接口实现

由接口生成子类不是通过 extends 实现的，而是用关键字 implements 来实现一个接口，方式为：

```
class 类名 implements 接口名 [extends 类名]
{
…//类体
}
```

其中,被实现的接口可以有多个,之间用逗号分隔。这一点是类在继承时是无法做到的。Java 只支持单继承,如果想达到多继承的效果,则可以通过多个接口的方式来实现。例如,根据上节定义的接口 Myinter,可以定义类 Test:

```
class Test implements Myinter
{
    public void show()                //实现抽象方法 show()
    {
        System.out.println("g="+g);
    }
}
```

与类之间的继承关系相似,接口之间也可以通过关键字 extends 进行继承,但其中抽象的方法不能被具体实现。

【例 5.11】 接口之间的继承示例。

```
package code0505;

interface IA {
    int a=1;
    void showa();
}
interface IB extends IA {                //接口 IB继承自 IA
    int b=2;
    void showb();
}
interface IC extends IA, IB {            //接口 C继承自 IA和 IB
    int c=3;
    void showc();
}
public class InterfaceTest implements IC {
    public void showa() {
        System.out.println("aaaa");
    }
    public void showb() {
        System.out.println("bbbb");
    }
    public void showc() {
        System.out.println("cccc");
    }
}
```

分析：由于InterfaceTest实现了接口IC，而IC又通过继承关系获取了抽象方法showa()、showb()和showc()，因此类InterfaceTest必须在实现代码中重写这3个抽象方法，缺一不可。

5.5.3 实用案例

【例5.12】 人驾驶不同汽车的多态实现。

假定一个司机拥有驾驶小汽车的执照，则通常情况下，他能驾驶宝马车就应该能驾驶奔驰车，甚至更多种类的车辆，因此，如何在车型可能不断增加、扩展的情况下，保持代码结构的稳定性是本例中除功能实现外更值得关心的问题。

程序代码如下：

```
package code0505.CarDrive;
public interface IDriver {
    public void drive(ICar car);     //这里接口 Idriver构成了对接口 Icar的依赖
}

package code0505.CarDrive;
public interface ICar {
    public void run();
}

package code0505.CarDrive;
public class BMW implements ICar {
    public void run() {
        System.out.println("宝马车正在运行…");
    }
}

package code0505.CarDrive;
public class Benz implements ICar {
    public void run() {
        System.out.println("奔驰车正在运行…");
    }
}
package code0505.CarDrive;
public class Driver implements IDriver {
    public void drive(ICar car) {    //参数为所有车型的父类 ICar
        car.run();
    }
}

package code0505.CarDrive;
public class Client {
    public static void main(String[] args) {
```

```
        IDriver ZhangSan=new Driver();
        ICar benz=new Benz();
        ZhangSan.drive(benz);          //开奔驰车,调用的是类 Benz 中的 run 方法
        ICar bmw=new BMW();
        ZhangSan.drive(bmw);           //开宝马车,调用的是类 BMW 中的 run 方法
        }
}
```

程序运行结果:

```
奔驰车正在运行…
宝马车正在运行…
```

分析:上述类和接口之间的关系如图 5-1 所示,接口 IDriver 构成了对接口 ICar 的依赖,Driver 是 IDriver 的类实现,BMW 和 Benz 是对 ICar 的类实现。

由于 Driver 类的 drive(ICar car)方法中使用了抽象的接口 ICar 作为形式参数,根据 5.3 节的运行多态规则,在向其传递实参时可以直接使用 ICar 的子类对象,如 Benz、BMW,当然如果 ICar 产生有新的车型(即新的子类)了,也可以直接通过 drive(ICar car)方法进行调用,这样调用的接口对用户是统一的,但实际调用的 run()方法根据传入参数是不同的。

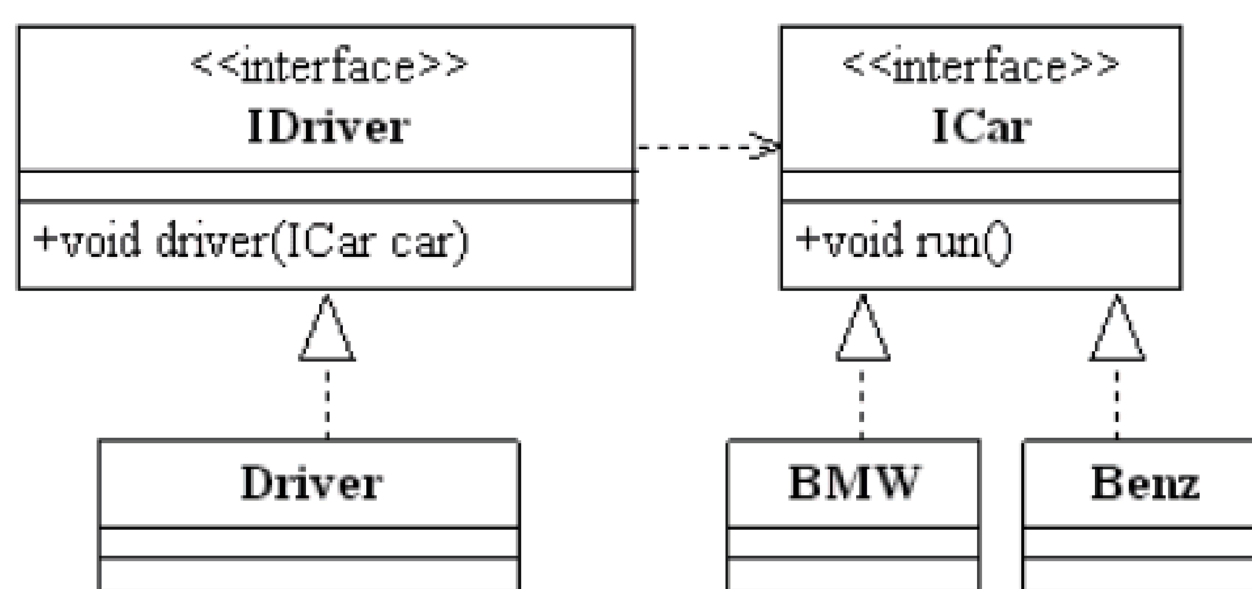


图 5-1　类人与汽车之间的关系图示

作为对比,如果将上例中的 Driver 类改为:

```
public class Driver {
    public void drive(Benz benz){       //参数类型为 Benz,只能向其传递 Benz 的对象
        benz.run();                      //调用 Benz 的 run 方法
    }
}
```

这时,Driver 类不再依赖于抽象的接口 ICar,而是直接依赖于具体的实现子类 Benz,这会给系统的稳定性造成影响,因为 public void drive(Benz benz)是专门针对 Benz 的,如果一个司机现在要驾驶 BMW 就需要修改 Driver 类,增加一个新的类似方法,如 public void drive(BMW bmw),如此反复,Driver 类的稳定性就会很差。

5.6　内　部　类

内部类是指在一个外部类的内部再定义一个类。内部类作为外部类的一个成员,依附于外部类而存在的。内部类主要有成员内部类、局部内部类、静态内部类和匿名内部类。

5.6.1 成员内部类

成员内部类作为外部类的一个成员存在,与外部类的属性、方法并列。

【例 5.13】 成员内部类示例。

```
package code0506;
public class Outer1 {
    private int k=20;
    public static void f1() {
    }

    public class Inner1 {
        public void f() {
            System.out.println("In inner Class");
        }
    }
}
```

编译上述代码会产生 Outer1.class 和 Outer$Inner1.class 两个文件。

5.6.2 局部内部类

在方法中定义的内部类称为局部内部类。与局部变量类似,局部内部类不能有访问控制符,因为它不是外部类的一部分,但是它可以访问当前代码块内的常量,和此外部类所有的成员。

【例 5.14】 局部内部类示例。

```
package code0506;
class Outer2 {
    public void doSomething() {
        class Inner2 {
            public void seeOuter() {
            }
        }
    }
}
```

5.6.3 静态内部类(嵌套类)

静态内部类又称静态嵌套类。静态的含义是该内部类可以像其他静态成员一样,没有外部类对象时也能够访问它。静态内部类不能访问外部类的成员和方法。

【例 5.15】 静态内部类示例。

```
package code0506;
class Outer3{
    static class Inner3{}
```

```
}
public class TestOuter3 {
    public static void main(String[] args){
        Outer3.Inner3 n=new Outer3.Inner3();
    }
}
```

5.6.4 匿名内部类

匿名内部类就是没有名字的内部类,简称匿名类。什么情况下需要使用匿名内部类?如果满足下面的一些条件,使用匿名内部类是比较合适的:

(1) 只用到类的一个实例。

(2) 类在定义后马上就用到。

(3) 类非常小。

由于匿名类没有名称,所以没办法引用它们,必须在创建时作为 new 语句的一部分来声明它们:

```
new<类或接口>  <类的主体>
```

这种形式的 new 语句声明一个新的匿名类,它继承某个给定的类,或者实现一个给定的接口,并创建该类的一个实例,并把它作为语句的结果返回。要继承的类或要实现的接口一般放在关键字 new 后边,然后是匿名类的主体定义。

【例 5.16】 匿名内部类示例。

```
package code0506;
interface Exam {
    void print1();
}
public class Outer4 {
    public Exam test() {
        return new Exam() {
            public void print1() {
                System.out.println("Hello world!!");
            }
        };
    }
    public static void main(String args[]) {
        Outer4 c=new Outer4();
        Exam e=c.test();
        e.print1();
    }
}
```

5.6.5 实用案例

【例 5.17】 匿名类在可视化界面设计中的应用。

如图 5-2 所示，创建一个可视化界面，界面中含有一个新建按钮，当用户单击该按钮时，弹出消息对话框，显示“单击了新建按钮”。



图 5-2　一个简单的可视化界面

Java 中匿名类用得最多的地方就是在可视化界面设计中，特别是将事件监听器注册到某个组件上的时候。

具体代码如下：

```
package code0506;
import javax.swing.* ;
import java.awt.event.* ;
public class QFrame extends JFrame {
    public QFrame() {
        JButton jbtNew=new JButton("New");
        JPanel panel=new JPanel();
        panel.add(jbtNew);
        add(panel);
        jbtNew.addActionListener(new ActionListener() {
        //新建一匿名类，并将该类对应的事件监听器注册到"新建"按钮 jbtNew 上
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(null, "单击了新建按钮");
                System.out.println("Process new");
            }
        });
    }
    public static void main(String[] args) {
        JFrame frame=new QFrame();
        frame.setTitle("QFrame");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setLocationRelativeTo(null);
        frame.pack();
        frame.setVisible(true);
    }
}
```

关于可视化界面设计与匿名类的更多内容，详见第 10 章。

5.7 Java 类的高级特性

5.7.1 泛型

1. 引入泛型的意义

Java 泛型编程是 JDK1.5 版本后引入的。泛型让编程人员能够使用类型抽象,通常用于集合里面。下面是一个不用泛型的简单例子:

```
List myIntList= new LinkedList();                //创建一个列表集合对象
myIntList.add(newInteger(0));                    //向列表集合中加入一个 Integer 对象
Integer x= (Integer)myIntList.iterator().next();
                                                 //通过 next 方法从该列表中取出存入对象
```

第 3 行代码令人困惑,因为程序员将 Integer 对象放入 List 中,但是在返回列表元素时,却要通过强制转换类型才能取得 Integer 对象,这是因为编译器只能保证 next()方法返回的是 Object 类型的对象,为保证 Integer 变量的类型安全,必须强制转换。

如果能保证列表中的元素为一个特定的数据类型,这样就可以取消类型转换,减少发生错误的机会,这也正是泛型设计的初衷。下面是一个使用了泛型的例子:

```
List< Integer> myIntList= newLinkedList< Integer> ();
myIntList.add(newInteger(0));
Integerx= myIntList.iterator().next();
```

在第 1 行代码中指定 List 中存储的对象类型为 Integer,这样在获取列表中的对象时,不必强制转换类型了。

2. 什么是泛型

泛型,即“参数化类型”。一提到参数,最熟悉的就是定义方法时要有形参,调用方法时传递实参。与之类似,也可以将“类型”定义成参数形式(称之为类型形参),然后在调用时传入具体的某种类型(称之为类型实参)。

例如,在类的方法 method(String str1,String str2)中,参数 str1、str2 的值通常是可变的,泛型也是一样,如下列泛型类定义 class Point＜T1,T2＞中,T1 和 T2 就像方法中的参数 str1 和 str2,也是可通过传递参数进行改变的。

3. 定义泛型类

【例 5.18】 泛型类定义和使用示例。

```
package code0507;
class Point< T1, T2> {
    T1 x;
    T2 y;
    public T1 getX() {
        return x;
    }
```

```
    public void setX(T1 x) {
        this.x=x;
    }
    public T2 getY() {
        return y;
    }
    public void setY(T2 y) {
        this.y=y;
    }
}

public class GeneDemo {
    public static void main(String[] args) {
        //实例化泛型类
        Point<Integer, Integer> p1=new Point<Integer, Integer>();
        p1.setX(10);
        p1.setY(20);
        int x=p1.getX();
        int y=p1.getY();
        System.out.println("This point is: "+x+", "+y);
        Point<Double, String> p2=new Point<Double, String>();
        p2.setX(25.4);
        p2.setY("东经 180 度");
        double m=p2.getX();
        String n=p2.getY();
        System.out.println("This point is: "+m+", "+n);
    }
}
```

程序运行结果：

```
This point is: 10, 20
This point is: 25.4, 东经 180 度
```

分析：与普通类的定义相比，上面的代码在类名后面多出了 <T1,T2>，T1 和 T2 是自定义的标识符，也就是上面提到的类型参数(用来传递数据的类型，而非数据的值)。T1 和 T2 只是数据类型的占位符，运行时会被替换为真正的数据类型。

泛型类在实例化时必须指出具体的类型，也就是向类型参数传值，格式为：

```
className<dataType1, dataType2> variable=new className<dataType1, dataType2>();
```

4. 定义泛型方法

除了定义泛型类，还可以定义泛型方法。

【例 5.19】 泛型方法定义和使用示例。

```
package code0507;
class NewPoint<T1, T2> {
```

```
        T1 x;
        T2 y;
        public T1 getX() {
            return x;
        }
        public void setX(T1 x) {
            this.x=x;
        }
        public T2 getY() {
            return y;
        }
        public void setY(T2 y) {
            this.y=y;
        }

        //定义泛型方法 printPoint
        public<T1, T2> void printPoint(T1 x, T2 y) {
            T1 m=x;
            T2 n=y;
            System.out.println("This point is: "+m+", "+n);
        }
    }
    public class GeneMethodDemo {
        public static void main(String[] args) {
            NewPoint<Integer, Integer> p1=new NewPoint<Integer, Integer>();
            p1.setX(10);
            p1.setY(20);
            p1.printPoint(p1.getX(), p1.getY());      //调用泛型方法 printPoint
            NewPoint<Double, String> p2=new NewPoint<Double, String>();
            p2.setX(25.4);
            p2.setY("东经 180 度");
            p2.printPoint(p2.getX(), p2.getY());      //调用泛型方法 printPoint
        }
    }
```

分析：上面的代码中定义了一个泛型方法 public<T1,T2>void printPoint(T1 x,T2 y),与传统方法相比,它在修饰符后、返回值类型前增加了类型参数(<>)。一旦定义了类型参数,就可以在参数列表、方法体和返回值类型中使用了。

与使用泛型类不同,使用泛型方法时不必指明参数类型,编译器会根据传递的参数自动查找出具体的类型。

提示：泛型方法与泛型类没有必然的联系,泛型方法可以有自己的类型参数,在普通类中也可以定义泛型方法。

5. 类型通配符

虽然 Integer 是 Object 的子类,但 Point<Object,Object>和 Point<Integer,Integer>

之间其实并没有什么关系，Point<Integer，Integer>也不是 Point<Object，Object>的子类。但有时需要一个在逻辑上可以同时表示 Point<Integer，Integer>和 Point<Double，Double>父类的一个引用类型，由此类型通配符应运而生。

类型通配符一般是使用问号"?"代替具体的类型实参。

注意：此处是类型实参，而不是类型形参。

【例 5.20】 类型通配符使用方法示例。

```
package code0507;
public class WildCardTest {
    public static void main(String[] args) {
        Box<String> name=new Box<String>("Hello");
        Box<Integer> age=new Box<Integer>(12);
        Box<Double> number=new Box<Double>(210.50);
        getData(name);
        getData(age);
        getData(number);
    }

    public static void getData(Box<?> data) {
        System.out.println("data:"+data.getData());
    }
}
class Box<T> {
    private T data;
    public Box() { }
    public Box(T data) {
        setData(data);
    }
    public T getData() {
        return data;
    }
    public void setData(T data) {
        this.data=data;
    }
}
```

程序运行结果：

```
data:Hello
data:12
data:210.5
```

5.7.2 Java 类加载机制

JVM 提供的运行时环境中有个模块是 ClassLoader(类加载器)，它主要用于将主类(即包含了 main 方法的类)加载到 JVM 的 code segment(代码区)，然后运行环境找到 main 方

法(程序入口)开始执行程序。在整个程序运行的过程中,逐步将更多的 class 动态加载到内存中,如图 5-3 所示。

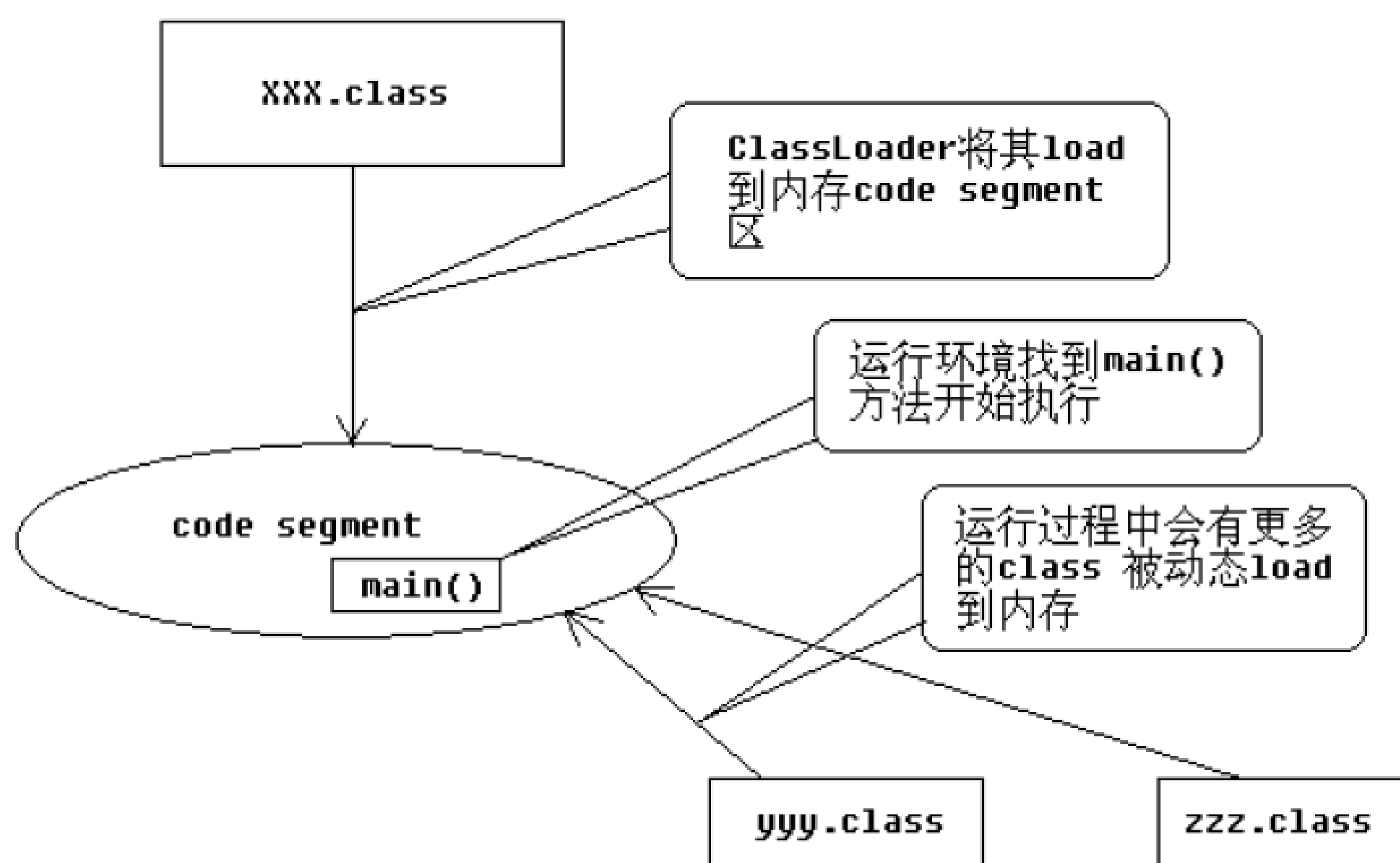


图 5-3　类的加载机制图示

利用 java -verbose:class XXX (XXX 为程序名)可以观察类的具体加载过程。

5.7.3　Java 反射机制

Java 反射机制是在运行状态中,对于任意一个类都能够知道这个类的所有属性和方法,对于任意一个对象都能够调用它的任意一个方法和属性,这种动态获取的信息以及动态调用对象的方法的功能称为 Java 的反射机制。正常方式与反射方式对比如图 5-4 所示。

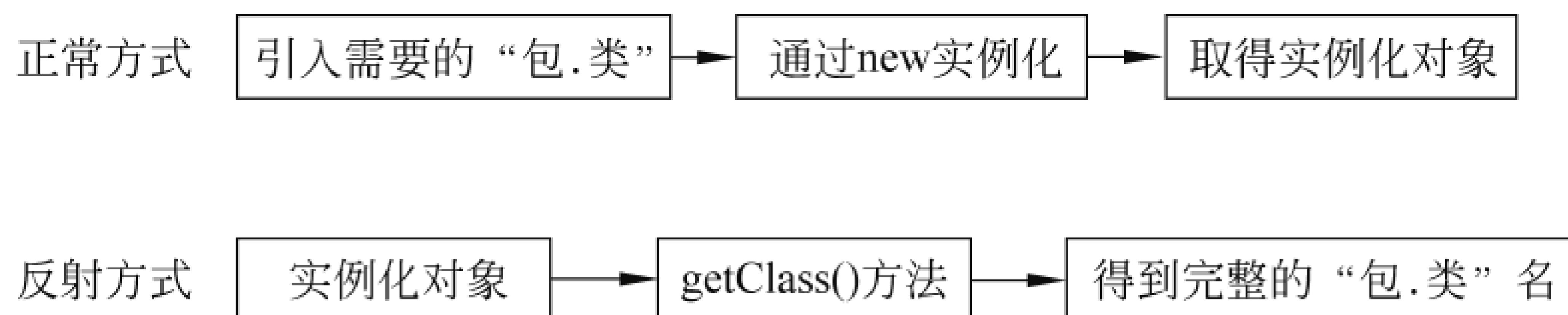


图 5-4　正常方式与反射方式对比

Java 的反射机制可以增加程序的灵活性,避免将程序写死到代码里。

例如,实例化一个 Person()对象,如果不使用反射,可以通过 new Person()实现,但如果想实例化其他类,就必须修改源代码,并重新编译。如果使用反射,则代码变为:Class.forName("Person").newInstance(),而且这个类描述还可以写到配置文件中,如**.xml,这样如果想实例化其他类,只要修改配置文件的类描述即可,不需要重新修改代码并编译。

Java 反射机制提供的主要功能包括:

(1) 加载运行时才能确定的数据类型。

(2) 解析类的结构、分析类的能力、获取其内部信息。

(3) 操作类或其实例(访问属性、调用方法、创建新对象)。

java.lang.Class 类是 Java 反射机制的基础,它是一个比较特殊的类,封装了被装入到 JVM 中的类(包括类和接口)的相关信息,可以通过它实现对相关类中详细信息的获取。

1. 获取 Class 类对象

一般可以通过以下 3 种方式获得 Class 类对象：

(1) 使用 Class 类的静态方法 forName，如对类 Person，Class.forName("Person")；。

(2) 使用对象的 getClass()方法，如 Person p=new Person()；Class c=p.getClass()；。

(3) 使用类名.class 方法，如 Class c3=Person.class；。

【例 5.21】 3 种获取 Class 类对象的方法。

```
package code0507;
public class GetClass {
    public static void main(String[] args) {
        checkClass();
    }
    public static void checkClass() {
        try {
            System.out.println("使用对象的 getClass 方法");
            Class cla=new Student().getClass();
            System.out.println(cla.getName());

            System.out.println("使用类名.class 方法");
            Class forClass=Student.class;
            System.out.println(forClass.getName());

            System.out.println("使用 Class 类的静态方法");
            Class forName=Class.forName("code0507.Student");
            System.out.println(forName.getName());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
class Student {
    private int id;
    private String name;
    public Student() {
        id=0;
        name="default";
    }
}
```

程序运行结果：

```
使用对象的 getClass 方法
code0507.Student
使用类名.class 方法
code0507.Student
使用 Class 类的静态方法
```

```
code0507.Student
```

2. 获取实例对象

不仅可以取得对象所在类的信息,也可以直接通过 Class 类的 newInstance 方法进行对象实例化操作。newInstance 方法原型如下:

```
public T newInstance() throws InstantiationException,IllegalAccessException
```

对例 5.18 中的类 Student,可以实例化对象如下:

```
Student s= (Student)Class.forName("code0507.Student").newInstance();
```

newInstance 方法调用默认的构造器初始化新创建的对象,如果这个类没有默认的构造器,那么就会抛出一个异常。

3. 获取类的构造方法

可以通过 Class 类中的 getConstructors()方法或 getDeclaredConstructors()方法获得本类中的全部构造方法,上述方法的返回类型都是 Constructor(构造器)的数组。

```
public Constructor<?>[]  getConstructors()
```

返回类中所有 public 的构造器集合。

```
public Constructor<T>  getConstructor(Class<?>…parameterTypes)
```

返回指定 public 构造器,参数为构造器参数类型集合。

```
public Constructor<?>[]  getDeclaredConstructors()
```

返回类中所有的构造器,包括私有的构造器。

```
public Constructor<T>  getDeclaredConstructor(Class<?>…parameterTypes)
```

返回任意指定的构造器。

【例 5.22】 获取 Person 类中的所有构造方法并实例化对象。

```
package code0507;
import java.lang.reflect.Constructor;
class Person {
    private String name;
    private int age;
    public Person() {
        this.name= "default";
    }
    public Person(String name) {
        this.name= name;
    }
    public Person(int age) {
        this.age= age;
    }
    public Person(String name, int age) {
        this.age= age;
```

```
            this.name=name;
        }
        public String getName() {
            return name;
        }
        public int getAge() {
            return age;
        }
        public String toString() {
            return "["+this.name+"  "+this.age+"]";
        }
    public int addAge(int a){
            return age+a;
        }
}

public class GetConstructor {
    public static void main(String[] args) {
        Class<?> demo=null;
        try {
            demo=Class.forName("code0507.Person");
        } catch (Exception e) {
            e.printStackTrace();
        }
        //取得全部的构造函数
        Constructor<?> cons[]=demo.getConstructors();
        try {
            Constructor cons0=demo.getConstructor();
            Constructor cons1=demo.getConstructor(String.class);
            Constructor cons2=demo.getConstructor(int.class);
            Constructor cons3=demo.getConstructor(String.class,int.class);
            Person per1=(Person) cons0.newInstance();
            Person per2=(Person) cons1.newInstance("ZhangShan");
            Person per3=(Person) cons2.newInstance(20);
            Person per4=(Person) cons3.newInstance("Lisi", 20);
            System.out.println(per1);//[default 0]
            System.out.println(per2);//[ZhangShan 0]
            System.out.println(per3);//[null 20]
            System.out.println(per4);//[Lisi 20]
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果：

```
[default  0]
[ZhangShan  0]
[null  20]
[Lisi  20]
```

提示：要想调用有参构造方法，必须使用 Constructor 类的 newInstance 方法，因为 Class 类的 newInstance 方法只能调用默认的无参构造方法。

4. 获取类的成员变量

成员变量用 Field 类进行封装，主要的方法包括：

```
public Field getDeclaredField(String name)
```

获取任意指定名字的成员。

```
public Field[] getDeclaredFields()
```

获取所有的成员变量。

```
public Field getField(String name)
```

获取任意 public 成员变量。

```
public Field[] getFields()
```

获取所有的 public 成员变量。

5. 获取类的成员方法

```
public Method[] getMethods()
```

获取所有的共有方法的集合。

```
public Method getMethod(String name,Class< ?> … parameterTypes)
```

获取指定公有方法 参数 1：方法名 参数 2：参数类型集合。

```
public Method[] getDeclaredMethods()
```

获取所有的方法。

```
public Method getDeclaredMethod(String name,Class< ?> … parameterTypes)
```

获取任意指定方法。

通过 Class 类的 getMethod 方法或 getDeclaredMethod 方法取得一个 Method 对象后，即可通过 invoke()方法调用指定的方法。invoke 方法结构如下：

```
public Object invoke(Object obj, Object… args)
```

其中，参数 obj 从中调用底层方法的对象；args 用于方法调用的参数。

【例 5.23】 通过反射机制获取类的属性和方法示例。

```
package code0507;
import java.lang.reflect.* ;
public class ClassDemo {
```

```
    public static void main(String args[]) {
        try {
            Class cls=Class.forName("code0507.Person");
            Constructor ct=cls.getConstructor(int.class);
            Object obj=ct.newInstance(20);
            //获取类中的所有方法及其属性
            Method[] methods=cls.getDeclaredMethods();
            for (Method m: methods) {
                System.out.print(Modifier.toString(m.getModifiers())+" "+m.getReturnType().getName
                ()+" "+m.getName()+"(");
                Class<?>[] paras=m.getParameterTypes();
                for (Class<?>p: paras) {
                    System.out.print(p.getName()+" ");
                }
                System.out.println(")");
            }

            Method meth=cls.getMethod("addAge", int.class);
            //调用指定的方法
            Object retObject=meth.invoke(obj, 5);
            System.out.println("After addAge():"+retObject);

            //获取类中所有的属性
            Field[] field=cls.getDeclaredFields();
            for (Field f: field) {
                //获得属性修饰符
                System.out.print(Modifier.toString(f.getModifiers())+" ");
                //获得属性类型
                System.out.print(f.getType().getName()+" ");
                //获得属性名称
                System.out.println(f.getName());
            }

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

程序运行结果：

```
public java.lang.String toString()
public java.lang.String getName()
public int addAge(int)
public int getAge()
After addAge():25
```

```
private java.lang.String name
private int age
```

如果想进一步获取类中方法的具体信息(如返回值类型、参数列表、修饰符),可以通过类 Method 中的下列方法实现。

取得全部的返回值:public Class<?>getReturnType()。

取得全部的参数:public Class<?>[] getParameterTypes()。

取得修饰符:public int getModifiers()。

同理,如果想进一步了解类中属性的具体信息(如修饰符、类型、属性名),可以通过类 Field 的下列方法实现。

取得属性修饰符:public int getModifiers()。

取得属性类型:public Class<?> getType()。

取得属性名称:public String getName()。

5.7.4 实用案例

【例 5.24】 对 List 对象按不同成员属性排序。

一般对 List 排序可以使用 Collections. sort(List),但如果 List 中包含的是一个对象,那么这种方法是行不通的。例如,给定对象 UserInfo 如下,现在要求编写一个通用的方法,可以任意指定成员属性,List 按该成员属性重新排序输出。

```
//类 UserInfo
public class UserInfo implements java.io.Serializable {
    private Integer userId;
    private String username;
    private java.util.Date birthDate;
    private Integer age;
    private SimpleDateFormat formater=new SimpleDateFormat("yyyy-MM-dd");
    public UserInfo() {
    }
    public UserInfo(Integer userId, String username, java.util.Date birthDate, Integer age) {
        this.userId=userId;
        this.username=username;
        this.birthDate=birthDate;
        this.age=age;
    }
    public void setUserId(Integer value) {
        this.userId=value;
    }
    public Integer getUserId() {
        return this.userId;
    }
    public void setUsername(String value) {
        this.username=value;
    }
```

```
    public String getUsername() {
        return this.username;
    }
    public void setBirthDate(java.util.Date value) {
        this.birthDate=value;
    }
    public java.util.Date getBirthDate() {
        return this.birthDate;
    }
    public void setBirthDatestr(String value) throws Exception {
        setBirthDate(formater.parse(value));
    }
    public String getBirthDatestr() {
        return formater.format(getBirthDate());
    }
    public void setAge(Integer value) {
        this.age=value;
    }
    public Integer getAge() {
        return this.age;
    }
    public String toString() {
         return new StringBuffer().append(getUserId()).append("; "+getUsername()).append("; "+
        getBirthDatestr())
                .append("; "+getAge()).toString();
    }
}
//类 MySort
package code0507.UserInfoSort;

import java.util.Collections;
import java.util.Comparator;
import java.util.List;
import java.lang.reflect.Method;
import java.lang.reflect.InvocationTargetException;
public class MySort<E> {
    public void Sort(List<E> list, final String method, final String sort) {
        Collections.sort(list, new Comparator() {
            public int compare(Object a, Object b) {
                int ret=0;
                try {
                    Method m1=((E) a).getClass().getMethod(method, null);
                    Method m2=((E) b).getClass().getMethod(method, null);
                    if (sort !=null && "desc".equals(sort))    //倒序
                        ret=m2.invoke(((E) b), null).toString().compareTo(m1.invoke(((E) a), null).
                        toString());
```

```
                        else                                    //正序
                            ret=m1.invoke(((E) a), null).toString().compareTo(m2.invoke(((E) b), null).
                            toString());
                    } catch (NoSuchMethodException ne) {
                        System.out.println(ne);
                    } catch (IllegalAccessException ie) {
                        System.out.println(ie);
                    } catch (InvocationTargetException it) {
                        System.out.println(it);
                    }
                    return ret;
                }
            });
        }
    }

    //测试类 Test
    package code0507.UserInfoSort;
    import java.util.ArrayList;
    import java.util.List;
    import java.text.SimpleDateFormat;
    public class Test {
        public static void main(String[] args) throws Exception {
            List<UserInfo> list=new ArrayList<UserInfo>();
            SimpleDateFormat formater=new SimpleDateFormat("yyyy-MM-dd");
            list.add(new UserInfo(3, "Zhang", formater.parse("2016-12-01"), 11));
            list.add(new UserInfo(1, "Li", formater.parse("2016-10-01"), 30));
            list.add(new UserInfo(2, "Chen", formater.parse("2015-10-01"), 11));
            MySort<UserInfo> sortList=new MySort<UserInfo>();
            //按 userId 排序
            sortList.Sort(list, "getUserId", "desc");
            System.out.println("--------按 userId 倒序------------------");
            for (UserInfo user: list) {
                System.out.println(user.toString());
            }
            //按 username 排序
            sortList.Sort(list, "getUsername", null);
            System.out.println("---------按 username 排序-----------------");
            for (UserInfo user: list) {
                System.out.println(user.toString());
            }
            //按 birthDate 排序
            sortList.Sort(list, "getBirthDatestr", null);
            System.out.println("---------按 birthDate 排序-----------------");
            for (UserInfo user: list) {
```

```
                System.out.println(user.toString());
            }
        }
    }
```

分析：排序类 MySort 中没有用到具体的对象和类型，而是使用了一个泛型 E，这使得该类及方法具有一定的通用性，如果要对 UserInfo 的 userId 排序，只需将方法名作为参数传入即可，以此类推。

进行排序时，通过 Collections.sort()方法来实现，其中 Comparator 是个比较器接口，可通过重写 compare()或 equals()方法实现比较功能。compare(a,b)方法可根据第一个参数小于、等于或大于第二个参数分别返回负整数、零或正整数。

【例 5.25】 使用反射类构造通用数据库查询应用。

假定数据库的每一个表对应一个 Java 类，表中的每一个字段对应该类中的一个属性，并且类的名字和表的名字相同，属性名和字段名也相同。如果数据库 blogsystem 中存在如图 5.5 所示的一张表 userinfo，则首先将表 userinfo 映射为对应的 Java 类，然后建立一个数据库工厂类 DBFactory 以实现对数据库的连接，最后利用反射类对作为传入参数的类进行解析，取得类名、属性名、类型等，并将其按照 SQL 语法规范拼接为一个完整的 SQL 语句，提交 DBMS 查询。

Column Name	Datatype	NOT NULL	AUTO INC	Flags
Id	INT(45)	✔		☑ UNSIGNED ☐ ZEROFILL
Name	VARCHAR(45)	✔		☐ BINARY
Pwd	VARCHAR(45)	✔		☐ BINARY
Age	INT(45)	✔		☑ UNSIGNED ☐ ZEROFILL

图 5-5　数据库 blogsystem 中的表 userinfo

```
//UserInfo类
package code0507.DB;
public class UserInfo {
    private int id;
    private String name;
    private String pwd;
    private int age;
    @Override
    public String toString() {
        return "UserInfo [id="+ id+ ", name= "+ name+ ", pwd= "+ pwd+ ", age= "+ age+ "]";
    }
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id= id;
    }
    public String getName() {
        return name;
```

```
        }
        public void setName(String name) {
            this.name=name;
        }
        public String getPwd() {
            return pwd;
        }
        public void setPwd(String pwd) {
            this.pwd=pwd;
        }
        public int getAge() {
            return age;
        }
        public void setAge(int age) {
            this.age=age;
        }
    }

    //DBFactory类
    package code0507.DB;
    import java.sql.Connection;
    import java.sql.DriverManager;
    public class DBFactory {
        public static Connection getDBConnection() {
            Connection conn=null;
            try {
                Class.forName("com.mysql.jdbc.Driver");
                String url="jdbc:mysql://localhost:3306/blogsystem";
                String user="root";
                String password="yourpass";
                conn=DriverManager.getConnection(url, user, password);
            } catch (Exception e) {
                e.printStackTrace();
            }
            return conn;
        }
    }

    //DBhandle类
    package code0507.DB;
    import java.lang.reflect.Field;
    import java.lang.reflect.Method;
    import java.sql.Connection;
    import java.sql.PreparedStatement;
    import java.sql.ResultSet;
```

```
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;

public class DBhandle {
        public static Object getObject(String className, int Id) {
        //得到表名字
        String tableName=className.substring(className.lastIndexOf(".")+1,
                className.length());
        //根据类名来创建 Class 对象
        Class c=null;
        try {
            c=Class.forName(className);
        } catch (ClassNotFoundException e1) {
            e1.printStackTrace();
        }
        //拼凑查询 sql 语句
        String sql="select * from "+tableName+" where Id="+Id;
        System.out.println("查找 sql 语句: "+sql);
        //获得数据库链接
        Connection con=DBFactory.getDBConnection();
        //创建类的实例
        Object obj=null;
        try {
            Statement stm=con.createStatement();
            //得到执行查寻语句返回的结果集,详见第 14 章 JDBC 技术
            ResultSet rs=stm.executeQuery(sql);
            //得到对象的方法数组
            Method[] methods=c.getMethods();
            //遍历结果集
            while(rs.next()) {
                obj=c.newInstance();
                //遍历对象的方法
                for (Method method: methods) {
                    String methodName=method.getName();
                    //如果对象的方法以 set 开头
                    if (methodName.startsWith("set")) {
                        //根据方法名字得到数据表格中字段的名字
                        String columnName=methodName.substring(3,methodName.length());
                        //得到方法的参数类型
                        Class[] parmts=method.getParameterTypes();
                        if (parmts[0]==String.class) {
                            //如果参数为 String类型,则从结果集中按照列名取得对应的值,并
                              且执行该 set 方法
```

```
                        method.invoke(obj, rs.getString(columnName));
                    }
                    if (parmts[0]==int.class) {
                        method.invoke(obj, rs.getInt(columnName));
                    }
                }

            }
        }

    } catch (Exception e) {
        e.printStackTrace();
    }
    return obj;
}
public static void main(String args[]) {
    //查找对象
    UserInfo userInfo= (UserInfo) getObject("code0507.DB.UserInfo", 6988);
    System.out.println("获取到的信息: "+userInfo);
}
}
```

注意:该案例中 DBhandle 类中的数据库查询方法是可重用的,只要用户提供对数据库表的任意映射类和 Id,并将其作为参数传入即可。

5.7.5 Java 注解

Java 从版本 5 开始引入了注解,现在很多 Java 框架中也在大量使用注解,如 Hibernate、Jersey、Spring。注解作为程序的元数据嵌入到程序当中,可以用一些解析工具或者编译工具进行解析。

注解是代码的一种附属信息,它遵循一个基本原则:注解不能直接干扰程序代码的运行,无论增加或删除注解,代码都能够正常运行,Java 语言解释器会忽略这些注解,而由第三方工具负责对注解进行处理。

注解的语法比较简单,除了@符号的使用以外,它基本上与 Java 的固有语法一致,Java 内置了 3 种注解,定义在 java.lang 包中。

(1) @Override:只能用在方法之上,用来告诉别人这一个方法是改写父类。

(2) @Deprecated:建议别人不要使用旧的 API 的时候使用,编译的时候会用来产生警告信息,可以设定在程序里的所有的元素上。

(3) @SuppressWarnings:表示关闭一些不当的编译器警告信息。

除使用内置注解外,程序员也可以自定义注解。创建自定义注解与创建接口相似,但是注解的 interface 关键字需要以@符号开头,基本格式如下:

```
public @interface 注解名 {定义体}
```

使用@interface自定义注解时，自动继承了java.lang.annotation.Annotation接口，由编译程序自动完成其他细节。自定义注解时，不能继承其他的注解或接口。下面是一个定义注解的实例。

```
package code0507;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    public int id();
    public String description() default "no description";
}
```

自定义注解时需要注意以下几点：

(1) 注解方法不能带有参数，也不能抛出异常。例如，boolean value(String str)、boolean value() throws Exception 等方式是非法的。

(2) 注解方法返回值类型限定为基本类型、String、Enums、Annotation或者是这些类型的数组。

(3) 注解方法可以有默认值。例如，String level() default "LOW_LEVEL"、int high() default 2 是合法的，也可以不指定默认值。

(4) 注解本身能够包含元注解，元注解被用来注解其他注解。

Java 5中定义了4个标准的元注解(meta-annotation)类型：@Target、@Retention、@Documented和@Inherited，它们可用于提供对其他annotation类型的说明。

① @Target表示该注解用于什么地方，可能的ElemenetType参数如下。

ElemenetType.CONSTRUCTOR：构造器声明。

ElemenetType.FIELD：域声明(包括enum实例)。

ElemenetType.LOCAL_VARIABLE：局部变量声明。

ElemenetType.METHOD：方法声明。

ElemenetType.PACKAGE：包声明。

ElemenetType.PARAMETER：参数声明。

ElemenetType.TYPE：类，接口(包括注解类型)或enum声明。

② @Retention表示在什么级别保存该注解信息。可选的RetentionPolicy参数如下。

RetentionPolicy.SOURCE：注解将被编译器丢弃。

RetentionPolicy.CLASS：注解在class文件中可用，但会被VM丢弃。

RetentionPolicy.RUNTIME：JVM将在运行时也保留注释，因此可以通过反射机制读取注解的信息。

③ @Documented将此注解包含在javadoc中。

④ @Inherited允许子类继承父类中的注解。

使用注解的基本语法为：

```
@<注解名> (<成员名 1>=<成员值 1>,<成员名 1>=<成员值 1>,…)
```

下面是在类 PasswordUtils 中使用注解 UseCase:

```
package code0507;
public class PasswordUtils {
    @UseCase(id=47, description="Passwords must contain at least one numeric")
    public boolean validatePassword(String password) {
        return (password.matches("\\w* \\d\\w* "));
    }
    @UseCase(id=48)
    public String encryptPassword(String password) {
        return new StringBuilder(password).reverse().toString();
    }
}
```

当然,使用注解最主要的部分还是在于对注解的处理,这就涉及注解处理器。从原理上讲,注解处理器就是通过反射机制获取被检查方法上的注解信息,然后根据注解元素的值进行特定的处理,从而产生不同的行为。

注解处理器使用 Java 中的反射机制来读取和分析被注解的源代码,使用的主要的包有 java.lang 和 java.lang.reflect。只有当一个 Annotation 类型被定义为运行时的 Annotation 后,该注解才能运行时可见,一旦该 class 文件被装载,保存在 class 文件中的 Annotation 就会被虚拟机读取。

AnnotatedElement 接口是所有程序元素(Class、Method 和 Constructor)的父接口,所以程序通过反射获取了某个类的 AnnotatedElement 对象之后,程序就可以调用该对象的如下 4 个方法来访问 Annotation 信息。

方法 1:

```
<T extends Annotation> T getAnnotation(Class<T> annotationClass)
```

返回该程序元素上存在的、指定类型的注解。

方法 2:

```
Annotation[] getAnnotations()
```

返回该程序元素上存在的所有注解。

方法 3:

```
boolean is AnnotationPresent(Class<?extends Annotation> annotationClass)
```

判断该程序元素上是否包含指定类型的注解,存在则返回 true,否则返回 false。

方法 4:

```
Annotation[] getDeclaredAnnotations()
```

返回直接存在于此元素上的所有注释。

【例 5.26】 注解处理器的简单示例。

```
package code0507;
import java.util.* ;
import java.lang.reflect.Method;
public class UserAnnotation {
    public static void main(String[] args) {
        List< Integer> li=new ArrayList< Integer> ();
        Collections.addAll(li, 47, 48, 49, 50);//将元素 47,48,49,50 插入集合 li 中
        trackUseCases(li, PasswordUtils.class);
    }
    public static void trackUseCases(List< Integer> li, Class< ?> cl) {
        for (Method m: cl.getDeclaredMethods()) {
            UseCase uc=m.getAnnotation(UseCase.class);
            if (uc !=null) {
                System.out.println("Found Use Case:"+uc.id()+" "+uc.description());
                li.remove(new Integer(uc.id()));
            }
        }
        for (int i: li) {
            System.out.println("Warning: Missing use case- "+i);
        }
    }
}
```

程序运行结果：

```
Found Use Case:47 Passwords must contain at least one numeric
Found Use Case:48 no description
Warning: Missing use case- 49
Warning: Missing use case- 50
```

5.8 继承与多态实训任务

【任务描述】

利用 Java 的多态性创建一个图形（Shape）类，并编程实现求正方形（Square）、三角形（Triangle）和圆（Circle）面积的功能。

【任务分析】

显然，本例至少需要定义 3 个类（Square、Triangle 和 Circle），它们虽然形状不同，但具有相似的行为：计算面积和周长等，因此可以先抽象出一个父类，并定义一个公共的接口，再在子类中重写计算面积的方法，以求得不同形状的面积。

【任务解决】

求正方形、三角形和圆的面积的程序代码如下：

```
/* 定义抽象图形类 Shape* /
```

```
abstract class Shape {
    public abstract double getArea();        //抽象方法取得图形的面积
}

/* 通过继承定义正方形子类 Square */
class Square extends Shape {
    private double height=0;                 //正方形的边长
    public Square(double height) {
        this.height=height;
    }

/* 重写 getArea()方法 */
    public double getArea() {
        return (this.height * this.height); //Math.pow(this.height, 2);
    }
}

/* 通过继承定义圆子类 Circle */
class Circle extends Shape {
    private double r=0;                      //圆的半径
    private final static double PI=3.14;
    public Circle(double r) {
        this.r=r;
    }
/* 重写 getArea()方法 */
    public double getArea() {
        return (PI * r * r);
    }
}

/* 通过继承定义三角形子类 Triangle */
class Triangle extends Shape {
    private double a=0;                      //三角形的边 1
    private double b=0;                      //三角形的边 2
    private double c=0;                      //三角形的边 3
    private double h=0;                      //三角形的高
    public Triangle(double a, double h) {
        this.a=a;
        this.h=h;
    }
    public Triangle(double a, double b, double c) {
        this.a=a;
        this.b=b;
        this.c=c;
```

```
    }
/* 重写 getArea()方法 */
    public double getArea() {
        if (h==0) {
            double s= (a+b+c)/2;
            return Math.pow(s* (s-a) * (s-b) * (s-c), 0.5);
                                                    //根据海伦公式求三角形的面积
        } else {
            return (a * h/2);
        }
    }
}

//测试类
public class TestShape{
    public static void main(String[] args) {
        Shape square=new Square(3);               //构造一个边长为 3 的正方形
        Shape circle=new Circle(2);               //构造一个半径为 2 的圆
        Shape triangle1=new Triangle(3, 4, 5);    //构造一个边长分别为 3,4,5 的三角形
        Shape triangle2=new Triangle(3, 4);       //构造一个高和底分别为 3,4 的三角形
        System.out.println(square.getArea());
        System.out.println(circle.getArea());
        System.out.println(triangle1.getArea());
        System.out.println(triangle2.getArea());
    }
}
```

习题与思考

1. 在 Java 语言中,下面关于类的继承关系描述正确的是(　　)。
   A. 一个子类可以有多个父类
   B. 一个父类可以有多个子类
   C. 子类可以使用父类的所有属性和方法
   D. 子类一定比父类有更多的成员方法

2. 编写一个类 Student,该类拥有属性:校名、学号、性别、出生日期。方法包含设置姓名和成绩(setName(),setScore()),再编写 Student 类的子类 Undergraduate(大学生)。Undergraduate 类除拥有父类的上述属性和方法外,还拥有附加的属性和方法:属性包括系(department)、专业(major);方法包含设置系别和专业(setDepartment(),setMajor())。

3. 现有以下接口的声明:

```
public interface CalcArea{
    double getArea();
```

```
}
```

(1) 定义圆类 Circle 并实现接口 CalcArea,圆类的主要成员变量为半径,构造方法的参数用于初始化半径。

(2) 定义矩形类 Rectangle 并实现接口 CalcArea,矩形类的主要成员变量包括长和宽,构造方法的参数用于初始化长和宽。

(3) 现有如下类 Tester,其功能是求存储在一个数组中的多个图形的面积之和,要求补全其中缺失的代码。

```
public class Tester{
  static __(1)__ shapes={ new Circle(1.0), new Rectangle(3.0, 4.0), new Circle(8.0) };
    public static void main(String[ ] args){
     System.out.println("total area="+ sumArea(shapes);
    }

    public static double sumArea(__(2)__ shapes){
      __(3)__
    }
}
```

4. 给出下列程序运行的结果:

```
public class Test extends TT{
    public static void main(String args[]){
    Test t=new Test("Tom");
    }
    public Test(String s){
        super(s);
        System.out.println("How do you do?");
    }
    public Test(){
        this("I am Tom");
    }
}
class TT{
    public TT(){
        System.out.println("What a pleasure!");
    }
public TT(String s){
    this();
    System.out.println("I am "+s);
    }
}
```

5. 抽象类可以实例化吗? 如何使用抽象类?

6. 定义一个泛型方法,把任意参数类型集合中的数据安全地复制到相应类型的数

组中。

7. 下列关于通过反射方式获取方法并执行的过程说法正确的是(　　)。

A. 通过对象名.方法名(参数列表)的方式调用该方法

B. 通过 Class.getMethod(方法名,参数类型列表)的方式获取该方法

C. 通过 Class.getDeclaredMethod(方法名,参数类型列表)获取私有方法

D. 通过 invoke(对象名,参数列表)方法来执行一个方法

第 6 章 Java 标准类库

本章学习目标

Java 系统中有很多预定义类，根据其功能不同划分成不同的包，所有包合称为类库。java.lang 和 java.util 包是应用最广泛的包，其中定义的很多类和接口是编程时经常用到的工具类。本章主要介绍这些常见类和接口的使用方法，主要内容包括：

(1) 字符串 String、StringBuffer 类的使用方法。

(2) 简单正则表达式的使用方法。

(3) 使用数据类型包装器进行数据类型转换。

(4) System 类和 Runtime 类的使用方法。

(5) 日期和日历类的使用方法。

(6) 常用集合类的使用方法。

6.1 Java 标准类库简介

Java 提供了有功能强大的类库，因而在进行 Java 应用开发时非常得心应手。例如，想比较两个字符串"abc"和"abd"的值是否相同，可以使用类 String 的方法 compareTo()，如果该方法的返回值为 0，则两个字符串相同。

用户程序在使用这些类之前，必须首先通过 import 语句引入这些类或所在的包。本章介绍的类主要来自下面两个包。

(1) java.lang 包。java.lang 包是 Java 语言的核心类库，包含了运行 Java 程序必不可少的系统类，例如基本数据类型、基本数学函数、字符串处理、线程和异常处理类等，这个包是由系统自动加载的。

(2) java.util 包。java.util 包中包含了 Java 语言中的一些实用工具，例如处理时间的 Date 类、集合类 ArrayList 等，使用它们开发者可以更方便快捷地编写程序。

6.2 字符串 String 类和 StringBuffer 类

Java 语言中字符串类主要包含初始化后不能改变的字符串类 String 和字符串内容可以动态改变的类 StringBuffer。

Java 将字符串作为对象来处理，在对象中封装了一系列的方法来进行字符串处理。利用 Java 字符串处理技术不仅可以减少程序设计的工作量，而且使程序编制更加规范，从而减少错误的发生。

字符串类还可以对正则表达式进行处理。正则表达式是一个字符串，用来描述字符串匹配的模式，可以使用正则表达式来进行字符串匹配、替换和分解。另外，在包 java.util.regex 提供了 Pattern 和 Matcher 两个类对正则表达式进行处理。

6.2.1 String 类

String 类位于 java.lang 包中，主要用来处理在初始化后其内容不能被改变的字符串。

1. 字符串对象的构造

字符串常量是用双引号括起来的一系列 Java 合法字符。在用赋值运算符进行字符串初始化时，JVM 自动为每个字符串生成一个 String 类的对象。例如，"Java is great"。

字符串变量的声明和其他类一样，格式如下：

```
String s;
```

(1) 调用 String 类的构造方法，使用字符串常量，构造新字符串对象。例如：

```
s=new String("We are students");
```

也可写成：

```
s="We are students";
```

声明和实例化对象也可一步完成：

```
String s=new String("We are students");
```

或

```
String s="We are students";
```

使用没有参数的构造方法可以生成一个空字符串对象。例如：

```
String s=new String();
```

表示 s 为空字符串，与空字符串" "等价。

(2) 可以由字符数组构造字符串。例如：

```
char cDem01[]={'2','3','4','5'};
char cDem02[]={'1','2','3','4','5'};
String strDem01=new String(cDem01);        //使用字符数组构造字符串
```

```
String strDem02=new String(cDem02,1,4);    //从字符数组的第 1个字符开始构造长度为 4
                                           //的字符串
```


利用上面的两个构造方法生成的字符串实例的内容均为"2345"。

(3) 可以由字节数组构造字符串,用于已知字符的编码构造字符串。例如:

```
byte cDem01[]={66,67,68};
byte cDem02[]={65,66,67,68};
String strDem01=new String(cDem01);        //使用字节数组构造字符串;
String strDem02=new String(cDem02,1,3);    //从字节数组的第 1个字节开始,取 3个字节构
                                           //造字符数组
```

利用上面的两个构造方法生成的字符串实例的内容均为"BCD"。

2. String 类的常用方法

String 类提供了 length()、charAt()、indexOf()、lastIndexOf()、getChars()、getBytes()和 toCharArray()等方法。在这些方法中,按用途可以分为字符串长度计算、字符串比较、字符串检索、字符串的截取和替换等方法,下面详细介绍这些方法。

1) 字符串长度计算

使用 String 类中的 length()方法可以获取一个字符串的长度,返回值为 int。例如:

```
String s="we are students",tom="我们是学生";
int n1,n2;
n1=s.length();                             //n1 的值是 15
n2=tom.length();                           //n2 的值 5
```

2) 字符串比较

字符串比较的方法有 equals()、equalsIgnoreCase()、startsWith()、endsWith()、regionMatches()、compareTo()和 compareToIgnoreCase()等方法。

equals(String s)方法用来比较当前字符串对象的内容是否与参数指定的字符串 s 的内容相同。例如:

```
String tom=new String("we are students");
String boy=new String("We are students");
String jerry=new String("we are students");
```

tom. equals(boy)的值是 false,tom. equals(jerry)的值是 true。

equalsIgnoreCase(String s)比较当前字符串对象是否与参数指定的字符串 s 相同,比较时忽略大小写。例如:

```
String tom=new String("ABC"),
Jerry=new String("abc");
```

tom. equalsIgnoreCase(Jerry)的值是 true。

3) startsWith()和 endsWith()方法

字符串对象调用 srartsWith(String s)方法,判断当前字符串对象的前缀是否是参数指定的字符串 s。例如:

```
String tom="220302620629021",jerry="21079670924022";
```

tom.startsWith("220")的值是 true,jerry.startsWith("220")的值是 false。

可以使用 endsWith(String s) 方法,判断一个字符串的后缀是否是字符串 s。例如:

```
String tom="220302620629021",jerry="21079670924022";
```

tom.endsWith("021")的值是 true,jerry.endsWith("021")的值是 false。

4) compareTo 和 compareToIgnoreCase 方法

compareTo(String s)方法,按字典顺序与参数 s 指定的字符串比较大小。如果当前字符串与 s 相同,则该方法返回值 0;如果当前字符串对象大于 s,则该方法返回正值;如果小于 s,该方法返回负值。例如:

```
String str="abcde";
str.compareTo("boy");         //小于 0
str.compareTo("aba")          //大于 0
str.compareTo("abcde")        //等于 0
```

按字典序比较两个字符串还可以使用 compareToIgnoreCase(String s)方法,该方法忽略大小写。

【例 6.1】 将字符串数组按字典序重新排列。

```
package code0602;
public class SortStrs{
  public static void main(String args[]){
    String a[]={"Java","Basic","C++","Fortran","SmallTalk"};
      for(int i=0;i<a.length-1;i++){
        for(int j=i+1;j<a.length;j++){
        if(a[j].compareTo(a[i])<0){
          String temp=a[i];
          a[i]=a[j];
          a[j]=temp;
        }
      }
    }
  for(int i=0;i<a.length;i++) {
      System.out.print(" "+a[i]);
    }
  }
}
```

程序运行结果:

```
Basic C++ Fortran Java SmallTalk
```

5) 字符串检索

搜索指定字符或者字符串在另一个字符串中出现的位置,可以用 indexOf()方法,定义如下:

```
public int indexOf(int ch)
public int indexOf(int ch,int fromIndex)
public int indexOf(String str)
public int indexOf(String str,int fromIndex)
```

一些重载方法可以通过 fromIndex 来指定匹配的起始位置。如果没有检索到字符或字符串,则返回值是－1。例如:

```
String strSource="I love Java";
int nPosition;
nPosition=strSource.indexOf('v');           //nPosition 的值为 4
nPosition=strSource.indexOf('a',9);         //nPosition 的值为 10
nPosition=strSource.indexOf("love");        //nPosition 的值为 2
nPosition=strSource.indexOf("love",0);      //nPosition 的值为 2
```

另外,还可以使用 lastIndexOf()方法搜索一个字符或者字符串在另外一个字符串中的最后位置,方法如下:

```
public int lastIndexOf(int ch)
public int lastIndexOf(int ch,int fromlndex)
public int lastIndexOf(String str)
public int lastIndexOf(String str,int fromIndex)
```

这些方法从后向前搜索,并且可以通过 fromIndex 来指定匹配的起始位置。如果没有检索到字符或字符串,则该方法返回的值是－1。例如:

```
String strSource="I love Java";
int nPosition;
nPosition=strSource.lastIndexOf('v');        //nPosition 的值为 9,最后一个
nPosition=strSource. lastIndexOf ('a',9);    //nPosition 的值为 8,倒数第二个
```

6) 字符串的截取

在字符串中截取子字符串时使用方法 substring(int beginIndex),该方法将获得一个当前字符串的子串,该子串是从当前字符串的 beginIndex 处截取到末尾所得到的字符串。

方法 substring(int beginIndex, int endIndex)获得的子串是从当前字符串的 beginIndex 处开始截取到 endIndex－1 结束所得到的字符串。

例如:

```
String strSource="Java is interesting";
String strNew1=strSource.substring(5); //strNew1="is interesting"
String strNew2=strSource.substring(5,6); //strNew2="i"
```

7) 字符串的替换

在 String 类中完成字符串替换的方法如下:

```
public String replace(char oldChar,char newChar)
```

用参数 newChar 指定的字符替换由 oldChar 指定的所有字符而得到的字符串。

```
public String replaceAll(String old,String new)
```

使用给定的子字符串 new 替换原字符串中所有匹配正则表达式 old 的子字符串。

```
public String trim()
```

去掉前后空白后的字符串。

例如：

```
String s="I mist theep ";
String temp=s.replace('t', 's');      //结果是"I miss sheep"
String s="   I am a student    ";
String temp=s.trim();                 //结果是"I am a student"
```

8）连接两个字符串

连接字符串的方法如下：

```
public String concat(String str)
```

该方法用于将两个字符串连接在一起，与字符串的＋操作符功能相同。

```
String strString=new String("01234");
String strAnothString="56789";
strString=StrString.concat(strAnotherString)//strString="0123456789"
```

【例 6.2】 String 类的使用。

```
package code0602;
public class UseString {
    public static void main(String args[]) {
        int n1, n2, n3;
        String vb="Visual Baisc", ja="java", s4="C++", s1, s2, s3;
        s1=vb.concat(ja);
        s2=s1.substring(8, 16);
        s3=vb.replace('a', 'x');
        n1=s1.length();
        n2=s1.indexOf(ja);
        n3=s1.lastIndexOf("Visual");
        System.out.println(s1);
        System.out.println(s2);
        System.out.println(s3);
        System.out.println(n1);
        System.out.println(n2);
        System.out.println(n3);
    }
}
```

程序运行结果：

```
Visual Baiscjava
```

```
aiscjava
Visuxl Bxisc
16
12
0
```

6.2.2 StringBuffer 类

StringBuffer 类表示的是一个本身内容可变的字符串对象,包含一个缓冲区,主要用于完成字符串的动态添加、插入和替换等操作。

1) 添加操作 append()

将一个字符添加到字符串的后面。在应用中,如果添加字符的长度超过缓冲区的长度,则缓冲区自动将长度进行扩充。

append 方法有多种重载形式,可以处理不同的数据类型。例如:

```
public StringBuffer append(boolean b)
public StringBuffer append(char c)
public StringBuffer append(char[] str)
```

可用来向字符串缓冲区添加布尔变量、字符、字符数组、双精度数、浮点数、整型数、长整型数以及对象类型的字符串和字符串等。

例如:

```
StringBuffer sbf=new StringBuffer("1+2=");
int i=3;
sbf.append(i);
System.out.println(sbf);
```

输出结果为:

```
1+2=3
```

2) 插入操作

下面为插入数据到字符串的方法 insert 的部分重载声明:

```
public StringBuffer insert(intoffset,Boolean b)
public synchronized StringBuffer insert(int offset,char[] str)
public synchronized StringBuffer insert(int offset,String str)
```

字符串缓冲区 StringBuffer 的插入操作主要用于动态地向 StringBuffer 中插入字符。offset 是插入的位置。根据方法中的参数类型,可以向字符串缓冲区插入逻辑变量、字符、字符数组、双精度数、浮点数、整型数、长整型数以及对象类型的字符串和字符串等。

例如:

```
StringBuffer sbf=new StringBuffer("1+=2");
int n=1;
sbf.insert(2,n);
System.out.println(sbf);
```

输出结果为：

```
1+1=2
```

3）删除字符

删除字符方法如下：

```
StringBuffer delete(int start,int end)
```

该方法用于删除字符串缓冲区中起始序号为 start、终止序号为 end－1 的字符。

下面的代码段为 delete 方法的例子：

```
StringBuffer sbfSource=new StringBuffer("You are the best");
sbfSource.delete(0,5);        //结果是 re the best
```

删除指定字符方法如下：

```
StringBuffer deleteCharAt(int index)
```

该方法用于删除字符串缓冲区中指定 index 位置的字符。

4）内容替换

字符串内容替换方法如下：

```
public StringBuffer replace(int start,int end,String str)
```

该方法将字符串缓冲区中起始位置为 start、终止位置为 end－1 的字符替换为由字符串 str 指定的内容。

下面的代码段为 replace 方法的例子：

```
StringBuffer sbfSource=new StringBuffer("You are the best!");
String str="I'm";
sbfSource.replace(0,7,str);        //结果为 I'm the best!
```

【例 6.3】 使用 StringBuffer 反转字符串。

```
package code0602;
public class ReverseString {
    public static void main(String args[]) {
        String strSource=new String("I love Java");
        StringBuffer sbuf=new StringBuffer(strSource);
        System.out.println(sbuf.reverse());      //调用 StringBuffer 的反转方法
        String strDest=reverseIt(strSource);
        System.out.println(strDest);
    }
    public static String reverseIt(String source) {
        int i, len=source.length();
        StringBuffer dest=new StringBuffer(len);
        for (i=(len-1); i>=0; i--)
            dest.append(source.charAt(i));
        return dest.toString();
```

```
        }
    }
```

程序运行结果：

```
avaJ evol I
avaJ evol I
```

上例编写了自己的反转方法，当然也可以调用 StringBuffer 类的 reverse()方法实现反转。

6.2.3 正则表达式

正则表达式是一个字符串，用来描述字符串匹配的模式，可以进行字符串匹配、替换和分解。

一个正则表达式是由普通的字符(如字符 a～z)以及特殊字符(元字符)组成的文字模式，用以描述在查找文字主体时待匹配的一个或多个字符串。正则表达式作为一个模板，将某个字符模式与所搜索的字符串进行匹配。例如，下面的正则表达式：

"java"：由普通字符组成。

"java.*"：包含有两个特殊字符.*，表示 java 后面跟有 0 个或者多个字符。

"[a-zA-Z]"：表示 a～z 或 A～Z 当中的任意一个字符。

1. 正则表达式示例

String 类的 matches()、split()、replaceFirst()和 replaceAll()方法都需要传入一个正则表达式作为参数。

【例 6.4】 使用多个条件分割字符串。

```
package code0602;
public class SplitString {
    public static void main(String[] args) {
        //构造包含逗号、句号的正则表达式
        String regex="[,。]+";
        //被分割的字符串
        String s="中国,你好。Java。。程序设计。";
        //使用逗号、句号分割字符串
        String[] ss=s.split(regex);
        for(String a:ss)
        {
            System.out.println(a);
        }
    }
}
```

程序输出结果：

```
中国
你好
```

本例构造了一个正则表达式进行字符串分割。正则表达式“[,。]+”表示任意一个或者多个逗号、句号。然后把正则表达式传递给 split()方法作为分隔符进行字符串分割。从输出结果可以看出,已成功地对字符串进行了分割。

2. 正则表达式的特殊字符

所谓特殊字符(元字符),就是一些有特殊含义的字符,要在正则表达式模式中包含元字符以使其不具有特殊含义,必须使用反斜杠(\)转义字符。正则表达式的特殊字符如表 6-1 所示。

表 6-1 正则表达式的特殊字符

字符	说　　明
$	匹配输入字符串的结尾位置。要匹配 $ 字符本身,请使用\ $
()	标记一个子表达式的开始和结束位置。子表达式可以获取供以后使用
*	匹配前面的子表达式零次或多次。要匹配 * 字符,请使用\ *
+	匹配前面的子表达式一次或多次。要匹配+字符,请使用\+
.	匹配除换行符\n 之外的任何单字符。要匹配.,请使用\
[	标记一个中括号表达式的开始。要匹配[,请使用\[
?	匹配前面的子表达式零次或一次,或指明一个非贪婪限定符。要匹配? 字符,请使用\?
\	将下一个字符标记为特殊字符或原义字符或向后引用或八进制转义符。例如,\n 匹配换行符。序列\\匹配\,而序列\(则匹配(
^	匹配输入字符串的开始位置,除非在方括号表达式中使用,此时它表示不接受该字符集合。要匹配^字符本身,请使用 \^
{	标记限定符表达式的开始。要匹配{,请使用\{
|	指明两项之间的一个选择。要匹配|,请使用\\|
\s	空格字符(空格键,tab,换行,换页,回车)
\S	非空格字符([\s])
\d	一个数字,在 Java 中使用\\d
\D	一个非数字的字符
\w	一个单词
\b	一个单词的边界
\G	前一个匹配的结束

除了这些特殊字符,还有限定符。限定符用来指定正则表达式的一个给定组件必须要出现多少次才能满足匹配。有 * 、+、?、{n}、{n,}、{n,m}共 6 种,其中,n 表示出现的最小次数,m 表示出现的最大次数,具体使用时需要替换成数字。

3. 正则表达式的字符类

使用方括号[和]定义字符类。可以使用字符类指定字符列表以匹配正则表达式中的一

个位置。例如,下面的正则表达式定义了匹配 bag、beg、big 和 bug 的字符类:

```
"b[aeiu]g"
```

使用连字符指定字符的范围。例如,A～Z、a～z 或 0～9。这些字符必须在字符类中构成有效的范围。例如,下面的字符类匹配 a～z 范围内的任何一个字符或任何数字:

```
"[a-z0-9]"
```

如果在字符类的开头使用尖号^字符,则将反转该集合的意义,即未列出的任何字符都认为匹配。下面的字符类匹配除小写字母(a～z)或数字以外的任何字符:

```
"[^a-z0-9]"
```

4. 正则表达式的工具类

为了更好地处理正则表达式,java. util. regex 包里面有两个工具类可以使用:Pattern 和 Matcher,它们的使用方法分为 3 步:

(1) 构造一个模式:Pattern p=Pattern. compile("[a-z]*");。

(2) 构造一个匹配器:Matcher m=p. matcher(str);str 是需要匹配的字符串。

(3) 进行匹配,得到结果:boolean b=m. matches();。

【例 6.5】 判断手机号码。

```
package code0602;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class JudgeMobileNumber {
    public static void main(String[] args) {
        Pattern p=null;
        Matcher m=null;
        boolean b=false;
        //正则表达式表示第一位是 1,第二位为 3,4,5,结尾为 9位数字的一串数字
        p=Pattern.compile("^[1][3-5]+\\d{9}");
        String[] numbers={ "13996332243", "1227788", "15676789065",
        "139abcd1234" };
        for (String s: numbers) {
            m=p.matcher(s);
            b=m.matches();
            System.out.println("手机号码正确: "+b);
        }
    }
}
```

程序的输出结果:

```
手机号码正确: true
手机号码正确: false
手机号码正确: true
手机号码正确: false
```

6.2.4 实用案例

【例 6.6】 使用正则表达式检查 IP 地址。

```
package code0602;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
public class CheckIPAddress {
    public static void main(String[] args) {
        Pattern p=null;
        Matcher m=null;
        boolean b=false;
        //构造匹配 IP地址的模式
        p=Pattern.compile("[1-2]\\d{0,2}+\\.[1-2]\\d{0,2}+\\.
[1-2]\\d{0,2}+\\.[1-2]\\d{0,2}");
        String[] ips={"192.168.1.1","192.168.1.1345",
"222.168.1.134","322.168.1.134"};
        for (String s: ips) {
            m=p.matcher(s);
            b=m.matches();
            System.out.println("IP正确:"+b);
        }
    }
}
```

程序输出结果:

```
IP正确:true
IP正确:false
IP正确:true
IP正确:false
```

分析:该案例的关键是构造一个检查 IP 地址的正则表达式。每个 IP 地址由 4 个字节组成,每个字节用 1~3 位十进制数字表示,并且第一位必须是 1 或 2。由于.在正则表达式中是特殊字符,必须进行转义。

6.3 数据类型包装器类

在 java.lang 包中有一些类和之前学过的基本数据类型有关,叫做包装器类(wrappers)。

Java 使用简单的数据类型,例如整型(int)和字符(char),这些数据类型不是对象层次结构的组成部分。它们通过值传递而不能直接通过引用传递给方法。有时需要对这些简单的类型建立对象表达式,例如,当把数据放到集合中时,需要首先将其包装成对象。为了能够用对象的形式代表简单数据类型,Java 提供了与每一种简单类型相应的类。本质上,这些类包装(wrap)了简单类型的数据。因此,通常将它们称为类型包装器或者包装器类。

每种简单类型都有对应的包装器类型，Integer、Long、Short、Byte、Double、Float、Character 和 Boolean 分别是简单类型 int、long、short、byte、double、float、char 和 boolean 对应的包装器类。

注意：包装器类的首字母是大写的。

下面以整型包装器类为例说明包装器类的使用。

6.3.1 整型包装器类

Byte、Short、Integer 和 Long 类分别是字节型(byte)、短整型(short)、整型(int)和长整型(long)整数类型的包装器。可以使用数字或者是能够转换为数字的字符串来创建它们的对象：

```
Integer oi=new Integer(1234);
Long   ol=new Long("1234");
```

另外，Java 还支持这些类型的自动装箱(Autobox)和拆箱(Unbox)。自动装箱就是自动地把简单类型封装成对应的对象类型，拆箱就是自动地把对象里面的值提取出来。例如：

```
Integer oi=1234;            //自动装箱
int i=oi;                   //自动拆箱
```

在这些类中定义了两个常量：MAX_VALUE 和 MIN_VALUE，分别表示每种数据类型允许的最大值和最小值。

这些类中还定义了以下一些类型的转换方法。

1. 基本类型到包装类型的转换

```
valueOf()
```

把简单类型转换为对象类型，把字符串类型解析为包装器类型。例如：

```
Integer oi=Integer.valueOf(1234);
Integer oi2=Integer.valueOf("1234");         //以十进制把字符串解析为数字
Integer oi3=Integer.valueOf("1234", 16);     //以十六进制把字符串解析为数字
```

2. 提取包装器里面的值

```
byteValue()
intValue()
longValue()
shortValue()
```

可以提取包装器里相应类型的值。例如：

```
int i=oi2.intValue();
byte b=oi2.byteValue();
```

3. 字符串到数字类型的转换

```
public static int parseInt(String s)              //将参数 string 解析为有符号的
                                                  //十进制整型 int 值,以下类似
public static long parseLong (String s)
```

```
public static byte parseByte(String s)
public static short parseShort(String s)
public static long parseInt(String s, int radix)      //按照指定的进制 radix 解析字符串
```

4. 数字类型到字符串的转换

```
String toString();
```

把对象里面的值按字面意思转成字符串。例如：

```
static String toString(int i, int radix)            //按照 radix 进制把整数 i 转换成字符串,转换
                                                    //结果含符号字符
static String toBinaryString(int value)             //按二进制转换成字符串,转换结果不含符
号字
                                                    //符,即最高位为 1 表示负数
static String toOctalString(int value)              //按八进制转换成字符串,转换结果不含符号
static String toHexString(int value)                //按十六进制转换成字符串,转换结果不含
符号
```

6.3.2 实用案例

程序设计中一个最常见的任务是将一个数字的字符串表达式转换成数字。幸运的是 Java 提供了一个方便的方法去完成这项任务。Byte、Short、Integer 和 Long 类分别提供了 parseByte()、parseShort()、parseInt()和 parseLong()方法。

以 Integer 为例,数字和字符串之间的转换,主要使用 parseInt()方法和 toString()方法。下面的例子使用 parseInt()方法将字符串转换成与之相应的整型(int)值。Integer 和 Long 类还同时提供了 toBinaryString()、toHexString()和 toOctalString()方法,可以分别将一个值以二进制、十六进制和八进制形式转化成字符串。

【例 6.7】 数字和字符串的转换。

```
package code0603;
public class Convert {
    public static void main(String[] args) {
        String s1="123456";
        String s2="56789";
        String s=null;
        int i1=0, i2=0, sum=0;
        //把字符串转换成整数
        i1=Integer.parseInt(s1);
        i2=new Integer(s2).intValue();
        sum=i1+i2;
        System.out.println("和: "+sum);
        //转换成二进制字符串形式
        String s3=Integer.toBinaryString(sum);
        System.out.println("二进制: "+s3);
        //转换成十六进制形式
```

```
            String s4= Integer.toHexString(sum);
            System.out.println("十六进制: "+ s4);
        }
    }
```


程序运行结果:

```
和: 180245
二进制: 101100000000010101
十六进制: 2c015
```

分析:从程序可以看出,把字符串转换成整数可以使用 parseInt()方法,也可以先从字符串构造 Integer 对象,然后提取整数值。

6.4 System 类和 Runtime 类

System 类包含了很多静态方法和变量,主要有标准输入(in)、输出(out)和错误输出(err)成员变量;对系统属性和环境变量的访问方法;加载文件和库的方法;还有快速复制数组的实用方法。

Runtime 类封装了系统运行时的环境,通过它可以控制 Java 虚拟机的状态和行为。

6.4.1 记录程序执行的时间

System 类的 currentTimeMillis()方法返回自 1970 年 1 月 1 日午夜起到现在的时间,时间单位是毫秒。如果要记录程序中一段程序的运行时间,可以在这段程序开始之前存储当前时间,在该段程序结束之际再次调用 currentTimeMillis()方法。执行该段程序所花费的时间为其结束时刻的时间值减去其开始时刻的时间值。

【例 6.8】 计算程序运行的时间。

```
package code0604;
public class RunElapsed {
    public static void main(String[] args) {
        long start, end, sum= 0, times= 1_000_000_000;
        System.out.println("执行 "+ times+ "次循环需要的时间");
        start= System.currentTimeMillis();
        for (int i= 0; i< times; i+ + ) {
            sum= sum+ i * i;
        }
        end= System.currentTimeMillis();
        System.out.println("需要的时间是: "+ (end- start)+ "毫秒");
    }
}
```

程序运行结果:

```
执行 1000000000 次循环需要的时间
```

需要的时间是：791毫秒

6.4.2 复制数组

使用 arraycopy()方法可以将一个任意类型的数组快速地从一个地方复制到另一个地方。这比使用 Java 中编写的循环要快得多。下面是一个用 arraycopy()方法复制两个数组的例子，将数组 a 复制给数组 b，然后又将数组 b 的部分元素复制给数组 a。arraycopy()函数的定义如下：

```
static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)
```

【例 6.9】 复制数组示例。

```
package code0604;
public class ArrayCopyDemo {
    public static void main(String[] args) {
        byte a[]={ 65, 66, 67, 68, 69, 70, 71 };
        byte b[]={ 88, 88, 88, 88, 88, 88, 88, 88, 88, 88 };
        System.out.println("a="+new String(a));
        System.out.println("b="+new String(b));
        System.arraycopy(a, 0, b, 0, a.length);
        System.out.println("b="+new String(b));
        System.arraycopy(b, 5, a, 0, 4);
        System.out.println("a="+new String(a));
    }
}
```

程序运行结果：

```
a=ABCDEFG
b=XXXXXXXXXX
b=ABCDEFGXXX
a=FGXXEFG
```

6.4.3 内存管理

可以使用 Runtime 类获取 Java 虚拟机内部的内存使用情况，但是不能直接创建 Runtime 的对象，需要使用它的静态方法 getRuntime()获取。

【例 6.10】 计算内存使用情况。

```
package code0604;
public class ShowMemory {
    public static void main(String[] args) {
        Runtime r=Runtime.getRuntime();
        System.out.println("Java 虚拟机总内存："+r.totalMemory());
        r.gc();//回收内存
        System.out.println("空闲内存 1:"+r.freeMemory());
```

```
            //创建 10000 个字符串对象
            String[] ss=new String[1000000];
            for(int i=0;i<ss.length;i++)
            {
                ss[i]=new String("Java 程序设计");
            }
            System.out.println("空闲内存 2:"+r.freeMemory());
            for(int i=0;i<ss.length;i++)
            {
                ss[i]=null;
            }
            r.gc();//回收内存
            System.out.println("空闲内存 3:"+r.freeMemory());
        }
    }
```

程序运行结果:

```
Java 虚拟机总内存:128974848
空闲内存 1:127743760
空闲内存 2:99566592
空闲内存 3:123761280
```

6.4.4 实用案例

可以使用 Runtime 类调用 Java 外部程序,例如,可以在 Java 程序执行的时候,使用 exec()方法调用操作系统的记事本程序。

【例 6.11】 打开记事本程序。

```
package code0604;
public class CallOsProgram {
    public static void main(String[] args) {
        Runtime r=Runtime.getRuntime();
        Process p=null;
        try {
            //打开记事本
            p=r.exec("notepad");
            //程序暂停 1秒
            Thread.sleep(1000);
            //关闭外部程序
            p.destroy();
        } catch (Exception e) {

            e.printStackTrace();
        }
    }
```

```
}
```

6.5 Math 类和 Random 类

6.5.1 Math 类

Math 类用于数学计算,包含了所有用于几何学、三角函数计算以及几种一般用途方法的浮点函数,这些函数都被定义为静态函数。

常用的三角函数及反三角函数,如正弦函数 static double sin(double arg)、反正弦函数 static double asin(double arg)等。

一些指数函数,如 static double pow(double y, double x),返回以 y 为底数、以 x 为指数的幂值。

其他函数,如绝对值函数 static int abs(int a)、最大值函数 static int max(int x, int y)、伪随机函数 static double random()等。

Math 类定义了两个双精度(double)常数:E(2.718281828459045)和 PI(近似为 3.141592653589793)。

下面通过一个例子来说明 Math 类的使用方法。

【例 6.12】 Math 类的使用。

```
package code0605;
public class MathDemo
{
    public static void main(String[] args)
    {
        double radians=2 * Math.PI;
        double d1=3.1415655678;
        System.out.println("ceil 函数:"+Math.ceil(d1));
        System.out.println("Round 函数:"+Math.round(d1));
        System.out.println("floor 函数:"+Math.floor(d1));
        System.out.println("exp 函数:"+Math.exp(d1));
        System.out.println("pow 函数:"+Math.pow(Math.E, d1));
        System.out.println("sin 函数:"+Math.sin(Math.PI/6));
        System.out.println("asin 函数:"+Math.asin(0.5));
        System.out.println("log 函数:"+Math.log(Math.E));
        System.out.println("角度转弧度: "+Math.toDegrees(radians));
        System.out.println("random 函数 1:"+Math.random());
        System.out.println("random 函数 2:"+Math.random());
    }
}
```

程序运行结果:

```
ceil 函数:4.0
```

```
Round函数:3
floor函数:3.0
exp函数:23.140065857331336
pow函数:23.140065857331333
sin函数:0.49999999999999994
asin函数:0.5235987755982989
log函数:1.0
角度转弧度: 360.0
random函数 1:0.26566448485431593
random函数 2:0.5993797846734983
```

6.5.2 Random 类

除了 Math 类中的 Random()方法可以获取随机数之外,在 Java 中还提供了一种可以获取随机数的类,即 java.util.Random 类。可以通过实例化一个 Random 对象创建一个随机数生成器。

```
Random random=new Random();
```

以这种形式实例化对象时,Java 编译器以系统当前时间作为随机数生成器的种子,因为每时每刻的时间不可能相同,所以产生的随机数也不同。如果运行速度太快,也会产生两次运行结果相同的随机数。

也可以在实例化 Random 类对象时,设置随机数生成器的种子。

```
Random random=new Random(seedValue);
```

Random 类中的关键方法如表 6-2 所示。

表 6-2 Random 类中的关键方法

方　法	说　明
nextInt()	返回一个随机整数
nextInt(int n)	返回大于等于 0、小于 n 的随机整数
nextLong()	返回一个随机长整型值
nextBoolean()	返回一个随机布尔型值
nextFloat()	返回一个随机浮点型值
nextDouble()	返回一个随机双精度型值
nextGaussian()	返回一个概率密度为高斯分布的双精度值

【例 6.13】 Random 类的使用。

```
package code0605;
import java.util.Random;
public class RandomDemo {
    public static void main(String[] args) {
```

```
        Random r=new Random();          //实例化一个 Random类
        System.out.println("随机产生一个整数: "+r.nextInt());
        System.out.println("随机产生一个大于等于 0且小于 10的整数: "+r.nextInt(10));
        System.out.println("随机产生一个布尔型的值: "+r.nextBoolean());
        System.out.println("随机产生一个双精度型的值 "+r.nextDouble());
        System.out.println("随机产生一个浮点型的值: "+r.nextFloat());
        System.out.println("随机产生一个概率密度为高斯分布的双精度值: "+r.
        nextGaussian());
    }
}
```

程序运行结果:

```
随机产生一个整数: 818664726
随机产生一个大于等于 0且小于 10的整数: 1
随机产生一个布尔型的值: false
随机产生一个双精度型的值: 0.18980303621562855
随机产生一个浮点型的值: 0.11750519
随机产生一个概率密度为高斯分布的双精度值: 1.1408564379397745
```

6.5.3 实用案例

【例 6.14】 随机生成字符数组并排序。

```
package code0605;
import java.util.Arrays;
import java.util.Random;
public class SortRandomArray {
    public static void main(String[] args) {
        char[] cc=new char[20];
        Random r=new Random();
        for(int i=0;i<cc.length;i++)
        {
            cc[i]=(char) ('A'+r.nextInt(26));
        }
        //排序前
        System.out.println(new String(cc));
        //排序
        Arrays.sort(cc);
        System.out.println(new String(cc));
    }
}
```

程序运行结果:

```
OXRBYODHDJTDJGWLGFSZ
BDDDFGGHJJLOORSTWXYZ
```

6.6　日期时间实用工具类

本节将介绍几个 java. util 包中的实用工具类，包括 Date、Calendar、Gregorian-Calendar 等。

6.6.1　Date(日期)

Date 类封装当前的日期和时间，也可以封装一个特定的日期。Date 支持下面的构造函数：

```
Date()
Date(long millisec)
```

第一种形式的构造函数用当前的日期和时间初始化对象；第二种形式的构造函数接收一个参数，该参数等于从 1970 年 1 月 1 日午夜起至今的毫秒数。Date 类中的方法如表 6-3 所示。

表 6-3　Date 类中的方法

方　　法	描　　述
boolean after(Date date)	如果调用对象所包含的日期迟于由 date 指定的日期，则返回 true；否则返回 false
boolean before(Date date)	如果调用对象所包含的日期早于由 date 指定的日期，则返回 true；否则返回 false
int compareTo(Date date)	将调用对象的值与 date 的值进行比较。如果两者数值相等，则返回 0；如果调用对象的值早于 date 的值，则返回一个负值；如果调用对象的值晚于 date 的值，则返回一个正值
boolean equals(Object date)	如果调用对象包含的时间和日期与由 date 指定的时间和日期相同，则返回 true；否则返回 false
long getTime()	返回自 1970 年 1 月 1 日起至今的毫秒数值
void setTime(long time)	按 time 的指定设置时间和日期，表示自 1970 年 1 月 1 日午夜至今的以毫秒为单位的时间值
String toString()	将调用 Date 对象转换成字符串并且返回结果

【例 6.15】　Date 类的使用。

```
package code0606;
import java.util.Date;
public class DateDemo
{
    public static void main(String[] args)
    {
        Date d1=new Date();
        Date d2=new Date(1140203030304L);
```

```
            System.out.println(d1);
            System.out.println(d2);
            if (d2.after(d1))
                System.out.println("d2 晚于 d1");
            System.out.println(d2.compareTo(d1));
            //改变的 d2 的值为 d1
            d2.setTime(d1.getTime());
            System.out.println(d2);
        }
    }
```

程序运行结果：

```
Fri Jun 17 10:12:34 CST 2016
Sat Feb 18 03:03:50 CST 2006
-1
Fri Jun 17 10:12:34 CST 2016
```

本例说明了如何构造日期对象、如何修改日期对象的值和进行日期比较。有 3 种方法可用于比较两个 Date 对象。首先，可以对两个对象使用 getTime()方法获得它们各自自 1970 年 1 月 1 日午夜起至今的毫秒数的值，然后比较这两个值的大小；其次，可以使用 before()、after()和 equals()方法；最后，可以使用 compareTo()方法，该方法实现了 Comparable 接口定义。

6.6.2 Calendar(日历)

抽象类 Calendar 提供了一组方法，这些方法允许将以毫秒为单位的时间转换为一组有用的分量：年、月、日、小时、分和秒。Calendar 定义了一些常量用于得到或设置日历分量：AM 表示上午，PM 表示下午，MONDAY 表示星期一，HOUR 表示小时，DATE 表示月中的天，DAY_OF_WEEK 表示星期中的天等。Calendar 的子类能提供具体的功能。

由 Calendar 类定义的一些常用的方法如表 6-4 所示。

表 6-4 Calendar 中常用的方法

方法	描述
abstract void add(int field, int amount)	将 amount 加到由 field 指定的时间或日期分量。为了实现减功能，可以加一个负数。field 必须是由 Calendar 定义的域之一，如 Calendar. HOUR
final int get(int field)	返回调用对象的一个分量的值。该分量由 field 指定
static Calendar getInstance()	对默认的地区和时区，返回一个 Calendar 对象
final Date getTime()	返回一个与调用对象的时间相等的 Date 对象
final void set(int field, int val)	在调用对象中，设置 field 分量为 val 指定的值
final void set(int year,int month,int day of month)	设置年月日

续表

方　　法	描　　述
final void set(int year,int month,int day of month,int hours,int minutes)	设置各种日期和时间分量
final void set(int year,int month,int day of month,int hours,int minutes,int seconds)	设置各种日期和时间分量
final void setTime(Date d)	设置为 Date 对象 d 中指定的日期

【例 6.16】 Calendar 类的使用。

```
package code0606;
import java.util.Calendar;
import java.util.GregorianCalendar;
public class CalendarDemo {
    public static void main(String[] args) {
        //获得日历对象
        Calendar c1=Calendar.getInstance();
        //设置一周的第一天为周一
        c1.setFirstDayOfWeek(Calendar.MONDAY);
        //输出日历
        System.out.println(c1.getTime().toString());
        //显示当前的日期的各个分量
        display(c1);
        //创建日历对象,时间为 2020 年 12 月 8 日晚上 8 时 8 分 8 秒
        Calendar c2=new GregorianCalendar(2020, 11, 8, 20, 8, 8);
        //计算今天距离 2010 年 12 月 8 日还有多少天
        int days=c2.get(Calendar.DAY_OF_YEAR)-c1.get(Calendar.DAY_OF_YEAR);
        System.out.println("距离 2020 年 12 月 8 日还有:"+days+"天");
        //设置日期和时间分量
        c1.set(2030, 11, 30);
        c1.set(Calendar.HOUR, 10);
        c1.set(Calendar.MINUTE, 29);
        c1.set(Calendar.SECOND, 22);
        System.out.println("更新后时间:");
        display(c1);
        //调整日期和时间
        c1.add(Calendar.DATE, 10);
        c1.add(Calendar.HOUR_OF_DAY, 10);
        System.out.println("调整后时间:");
        display(c1);
    }
    static void display(Calendar c) {
        String months[]={"一月","二月","三月","四月","五月","六月","七月","八月","九月",
        "十月","十一月","十二月"};
```

```
        String days[]= { "", "星期一", "星期二", "星期三", "星期四", "星期五", "星期六", "星期日"
        };
        System.out.print("日期: ");
        //下面的代码打印各个分量的值
        System.out.print(c.get(Calendar.YEAR)+"年");
        System.out.print(months[c.get(Calendar.MONTH)]);
        System.out.print(c.get(Calendar.DATE)+"日 ");
        System.out.println(days[c.get(Calendar.DAY_OF_WEEK)-1]);
        System.out.print("时间: ");
        System.out.print(c.get(Calendar.HOUR_OF_DAY)+":");
        System.out.print(c.get(Calendar.MINUTE)+":");
        System.out.println(c.get(Calendar.SECOND));
    }
}
```

程序运行结果:

```
日期: 2016年六月 17日 星期五
时间: 10:27:19
距离 2020年 12月 8日还有: 174天
更新后时间:
日期: 2030年十二月 30日 星期一
时间: 10:29:22
调整后时间:
日期: 2031年一月 9日 星期四
时间: 20:29:22
```

6.6.3 实用案例

【例 6.17】 日期的格式化。

日历 Calendar 和 Date 的字符串显示默认为英文格式,为了显示成中文格式,上例创建了自定义函数。现在介绍一个工具类 java.text.SimpleDateFormat 格式化日期,即把日期对象转换为特定的字符串形式。

```
package code0606;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.Calendar;
import java.util.Date;
public class FormatDate {
    public static void main(String[] args) throws ParseException {
        //创建格式化日期,年月日等使用特定字母作为占位符
        SimpleDateFormat myFmt=new SimpleDateFormat("yyyy年 MM月 dd日
        HH时 mm分 ss秒");
        SimpleDateFormat myFmt1=new SimpleDateFormat("yy/MM/dd HH:mm");
        SimpleDateFormat myFmt2=new SimpleDateFormat("yyyy-MM-dd
```

```
            HH:mm:ss");
            SimpleDateFormat myFmt3=new SimpleDateFormat("yyyy年 MM月 dd日
            HH时 mm分 ss秒 E ");
            SimpleDateFormat myFmt4=new SimpleDateFormat("一年中的第 D天
            一年中第 w个星期 一月中第 W个星期 在一天中 k时 z时区");
            Date now=new Date();
            System.out.println(now.toString());
            //格式化输出
            String ds1=myFmt.format(now);
            System.out.println(ds1);
            System.out.println(myFmt1.format(now));
            System.out.println(myFmt2.format(now));
            System.out.println(myFmt3.format(now));
            System.out.println(myFmt4.format(now));
            //解析字符串为日期对象
            String ds2="2026年 06月 17日 10时 38分 52秒";
            Date d=myFmt.parse(ds2);
            //创建日历对象,日期计算
            Calendar c=Calendar.getInstance();
            c.setTime(d);
            c.add(Calendar.YEAR, 20);
            System.out.println(c.getTime());
        }
    }
```

程序运行结果:

```
Fri Jun 17 10:46:18 CST 2016
2016年 06月 17日 10时 46分 18秒
16/06/17 10:46
2016-06-17 10:46:18
2016年 06月 17日 10时 46分 18秒 星期五
一年中的第 169天 一年中第 25个星期 一月中第 3个星期 在一天中 10时 CST时区
Sun Jun 17 10:38:52 CST 2046
```

6.7 Java 集合类

与数组类似,Java 提供集合类的主要目的在于“保存多个数据/对象”,提供一种比数组下标存取方式更灵活的方法。

集合类存在于 java.util 包中,主要分为 3 种:Set、List 和 Map。Set 表示不允许容纳重复元素的集合;List 表示可以容纳重复元素的集合;Map 表示存储键/值对的集合,每个键/值对称为一项。

6.7.1 集合接口

集合框架定义的几个接口,决定了集合框架各类的基本特性。不同的是,具体类仅仅是

提供了标准接口的不同实现。支持集合的主要接口如表 6-5 所示。

表 6-5　支持集合的主要接口

接　　口	描　　述
Collection	集合框架的顶层接口,定义了操作对象集合的共同方法
List	继承 Collection,表示有序的、可包括重复元素的列表
Set	继承 Collection,表示无序的、无重复元素的集合(数学上的含义)
SortedSet	继承 Set,对 Set 中的元素进行排序

1. Collection 接口

Collection 接口是构造集合框架的基础,它声明所有集合类都拥有的核心方法,如表 6-6 所示。熟悉这些方法对于清楚地理解框架是十分必要的。

表 6-6　Collection 定义的主要方法

方　　法	描　　述
boolean add(Object obj)	将 obj 加入到类集合中,如果操作成功则返回 true;如果操作失败,则返回 false
boolean addAll(Collection c)	将 c 中的所有元素都加入到类集合中,如果操作成功,则返回 true;否则返回 false
void clear()	删除所有元素
boolean contains(Object obj)	如果 obj 是调用类集合的一个元素,则返回 true;否则返回 false
boolean equals(Object obj)	如果类集合与 obj 相等,则返回 true;否则返回 false
boolean isEmpty()	如果类集合是空的,则返回 true;否则返回 false
Iterator iterator()	返回类集合的迭代器
boolean remove(Object obj)	删除 obj,若成功删除则返回 true;否则返回 false
boolean removeAll(Collection c)	删除 c 的所有元素。若成功,则返回 true;否则返回 false
boolean retainAll(Collection c)	删除除包含在 c 中的元素之外的全部元素
int size()	返回类集合中元素的个数
Object[] toArray()	把集合转换为数组

2. List 接口

List 是有序的 Collection,使用此接口能够精确地控制每个元素插入的位置。用户能够使用索引(元素在 list 中的位置,类似于数组下标)来访问 List 中的元素,这类似于 Java 的数组。一个列表可以包含重复元素。

3. Set 接口

Set 是一种不包含重复元素的 Collection,即任意的两个元素 e1 和 e2 都有:e1. equals(e2)==false,Set 最多有一个 null 元素,它本身并没有定义任何附加的方法。

SortedSet 接口继承了 Set,并说明了元素按序排列的集合的特性。

6.7.2　实现 List 接口的类

实现 List 接口的类如图 6-1 所示，其中用斜体表示抽象类，一般不能直接使用，可以直接使用的类是 ArrayList、LinkedList、Vector 和 Stack。这几种类虽然在内部实现上有所不同，但由于实现了相同的接口，其使用方式基本相同。下面以 ArrayList 为例讲解其使用方法。

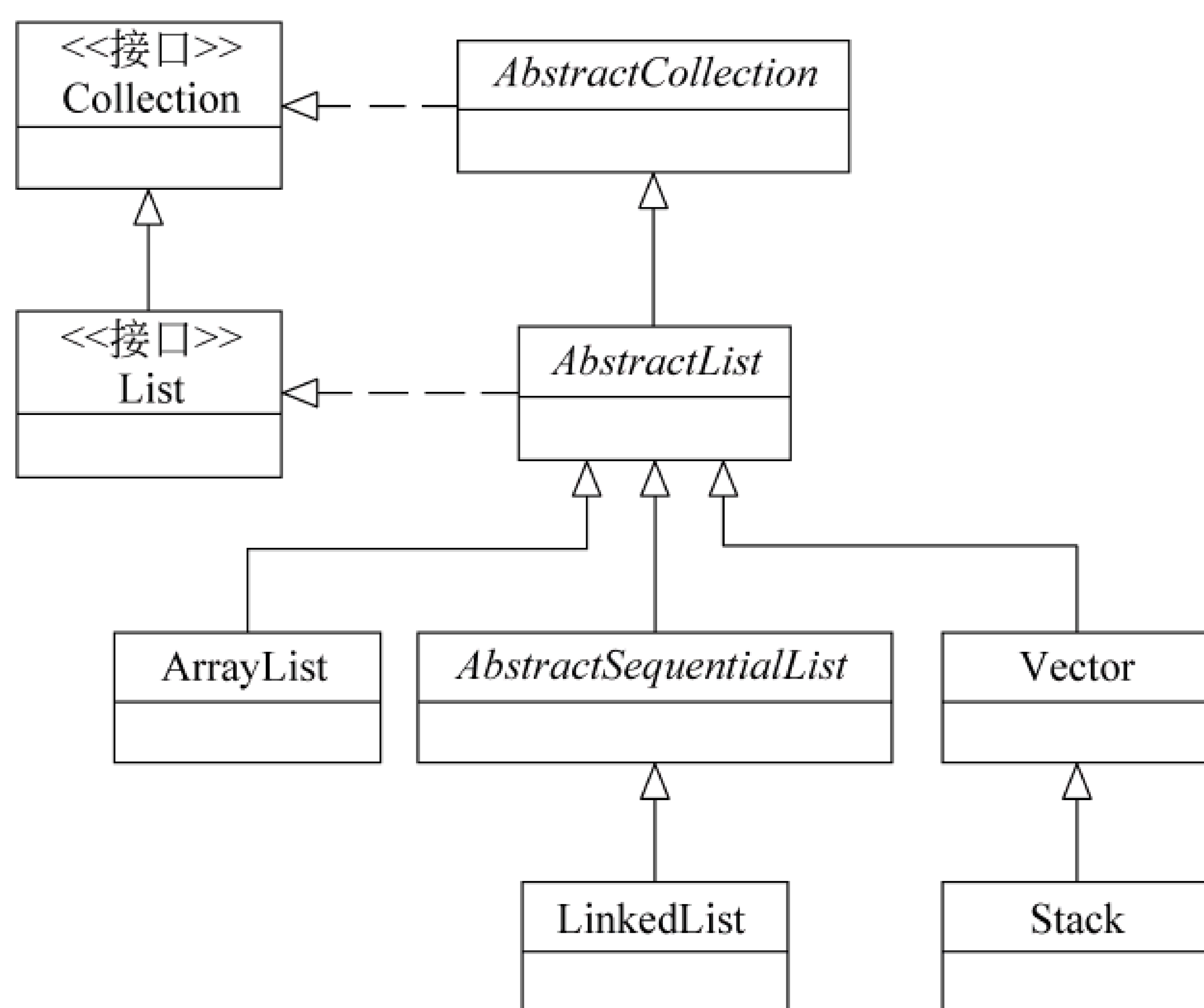


图 6-1　实现 List 接口的类

1. ArrayList 类

ArrayList 类支持可随需要而增长的动态数组。在 Java 中，标准数组是定长的，在数组创建之后，它们不能被加长或缩短。ArrayList 类能够动态地增加或减小其大小。ArrayList 类的对象以一个初始大小被创建，当超过了它的大小，可以自动增大；当对象被删除后，可以自动缩小。

ArrayList 构造函数如下：

```
ArrayList()
ArrayList(Collection c)
ArrayList(int capacity)
```

其中，第一个构造函数建立一个空的 ArrayList；第二个构造函数由集合 c 中的元素初始化。第三个构造函数指定初始容量(capacity)，容量是用于存储元素的基本数组的大小。当元素被追加到数组列表上时，容量会自动增加。

ArrayList 中提供的主要方法如下。

boolean add(Object o)：将指定的元素追加到此列表的尾部。

void add(int index, Object o)：将指定的元素插入列表的指定位置，index 从 0 开始。

void clear()：移除此列表中的所有元素。

boolean contains(Object o)：如果此列表中包含指定的元素，则返回 true。

Object get(int index)：返回此列表中指定位置上的元素。

boolean isEmpty()：测试此列表中是否为空。

Object remove(int index)：移除此列表中指定位置上的元素。

boolean remove(Object o)：从此列表中移除指定元素。

int size()：返回此列表的元素个数。

【例 6.18】 ArrayList 类的使用。

```
package code0607;
import java.util.ArrayList;
public class ArrayListDemo {
    public static void main(String[] args) {
        //创建一个 List
        ArrayList al=new ArrayList();
        System.out.println("List 的初始大小："+al.size());
        //向 List 中增加元素
        al.add("A");
        al.add("B");
        al.add("C");
        al.add("D");
        al.add("E");
        al.add("F");
        al.add(1, "A2");
        System.out.println("增加元素后的大小："+al.size());
        //显示其内容
        System.out.println("List 中的内容："+al);
        //删除 List 中的元素
        al.remove("D");
        al.remove(2);
        System.out.println("删除元素后的大小："+al.size());
        System.out.println("内容："+al);
        //转化为数组
        Object[] aa=al.toArray();
        for(Object a:aa)
        {
            System.out.print(a);
        }
    }
}
```

程序运行结果：

```
List 的初始大小：0
增加元素后的大小：7
List 中的内容：[A, A2, B, C, D, E, F]
删除元素后的大小：5
内容：[A, A2, C, E, F]
AA2CEF
```

2. Vector 类与 Stack 类

Vector 类非常类似于 ArrayList,但是 Vector 类是同步的,在多个线程同时访问时安全性更好,但这也使得它的性能稍差。

Stack 类继承自 Vector 类,实现一个后进先出的栈。Stack 类提供 5 个额外的方法使得 Vector 类可以当作栈使用。

boolean empty():测试栈是否为空。

peek():查看栈顶部的对象,但不从堆栈中移除它。

pop():移除栈顶部的对象,返回该对象。

push(E item):把项压入栈顶部。

int search(Object o):返回对象在栈中的位置,以 1 为基数。

6.7.3 实现 Set 接口的类

实现 Set 接口的类如图 6-2 所示。其中用斜体表示抽象类,一般不能直接使用,可以直接使用的类是 HashSet、TreeSet、LinkedHashSet。这几种类虽然在内部实现上有所不同,但由于其实现了相同的接口,因此使用方法基本相同。TreeSet 另外实现了 SortedSet 接口,可以对集合中的元素排序。下面以 HashSet 和 TreeSet 为例进行介绍。

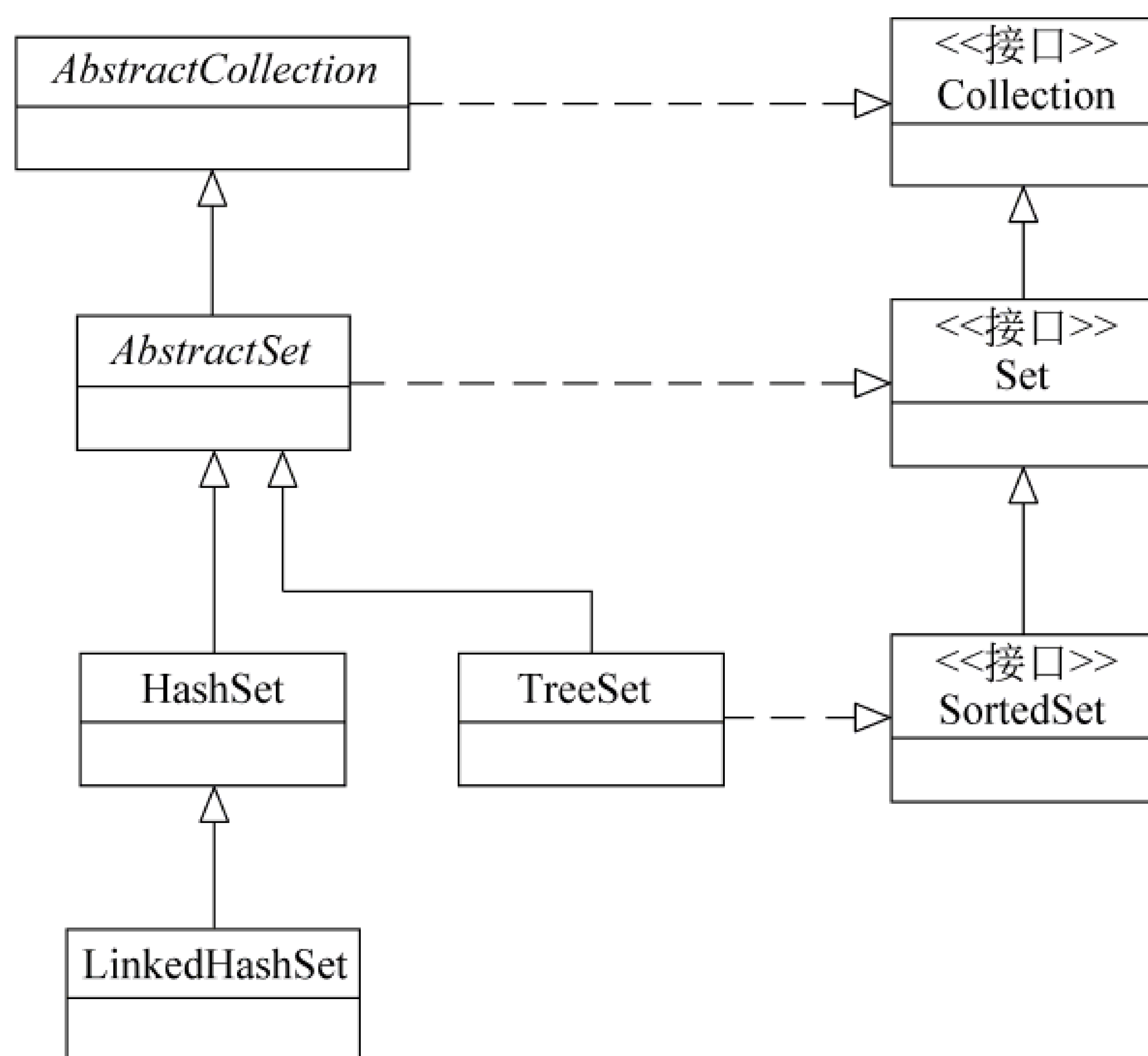


图 6-2 实现 Set 接口的类

1. HashSet 类

HashSet 类继承 AbstractSet 并且实现 Set 接口。它创建了一个类集合,该类集合使用散列表进行存储。散列表通过使用散列法的机制来存储信息。

在散列(hashing)中,一个关键字被用来确定唯一的一个值,称为散列值(hashcode)。而散列值被用来当作与关键字相连的数据的存储下标。关键字到其散列值的转换是自动执行的——看不到散列值本身,程序代码也不能直接索引散列表。

HashSet 类的常用构造函数如下:

```
HashSet()
```

```
HashSet(Collection c)
```

第一种形式构造一个默认的散列集合;第二种形式用集合 c 中的元素初始化散列集合。

此外,HashSet 提供下列方法对集合元素进行各种操作。

boolean add(Object o):添加元素。

void clear():从此集合中移除所有元素。

boolean contains(Object o):如果此集合不包含指定元素,则返回 true。

boolean isEmpty():如果此集合不包含任何元素,则返回 true。

boolean remove(Object o):移除元素。

int size():返回此集合中的元素的数量。

提示:散列集合并没有确保其元素的顺序。如果需要排序存储,另一种类集——TreeSet 将是一个更好的选择。

2. TreeSet 类

TreeSet 类是使用树结构存储元素的类集合,对象按升序存储。访问和检索速度很快。在存储了大量的需要进行快速检索的排序信息的情况下,TreeSet 是一个很好的选择。

除包括常见的 add()、clear()、remove()、contains()等方法外,根据对有序元素访问的需要,TreeSet 还提供下列方法。

Object first():返回第一个(最低)元素。

Object last():返回最后一个(最高)元素。

Object floor(Object e):返回小于等于给定元素的最大元素;如果不存在,则返回 null。

Object higher(Object e):返回大于给定元素的最小元素;如果不存在,则返回 null。

Object lower(Object e):返回小于给定元素的最大元素;如果不存在,则返回 null。

Object pollFirst():获取并移除第一个(最低)元素;如果集合空,则返回 null。

Object pollLast():获取并移除最后一个(最高)元素;如果集合空,则返回 null。

【例 6.19】 Set 的使用。

```
package code0607;
import java.util.HashSet;
import java.util.Random;
import java.util.TreeSet;
public class SetDemo {
    public static void main(String[] args) {
        TreeSet ts=new TreeSet();
        HashSet hs=new HashSet();
        Random r=new Random();
        for (int i=0; i<10; i++) {
            char c=(char) ('A'+r.nextInt(25));
            ts.add(c);
            hs.add(c);
        }
        System.out.println("HashSet:"+hs);
        System.out.println("TreeSet:"+ts);
```

```
            //操作 TreeSet
            System.out.println(ts);
            System.out.println(ts.lower('T'));
            System.out.println(ts.pollFirst());
            System.out.println(ts.pollLast());
            System.out.println("TreeSet:"+ts);
            //操作 HashSet,保留和 ts 相同的元素
            hs.retainAll(ts);
            System.out.println("HashSet:"+hs);
        }
    }
```

程序运行结果:

```
HashSet:[Q, B, D, U, E, W, H, K]
TreeSet:[B, D, E, H, K, Q, U, W]
[B, D, E, H, K, Q, U, W]
Q
B
W
TreeSet:[D, E, H, K, Q, U]
HashSet:[Q, D, U, E, H, K]
```

分析: 从输出结果可以看出,TreeSet 按树存储其元素,元素按照自然顺序自动排序,也不能有重复元素。

6.7.4 通过迭代接口访问集合类

依次访问集合中的元素最简单的方法是使用迭代器,迭代器是一个实现 Iterator 或者 ListIterator 接口的对象。Iterator 可以遍历集合中的元素,从而获得或删除元素。ListIterator 继承 Iterator,允许双向遍历列表,并可以修改。Iterator 接口中的主要方法如表 6-7 所示,ListIterator 接口中的主要方法如表 6-8 所示。

表 6-7 Iterator 接口中的主要方法

方　　法	描　　述
boolean hasNext()	如果存在更多的元素,则返回 true;否则返回 false
Object next()	返回下一个元素
void remove()	从集合中删除当前元素

表 6-8 ListIterator 接口中的主要方法

方　　法	描　　述
void add(Object obj)	将 obj 插入列表中的一个元素之前,该元素在下一次调用 next()方法时返回
boolean hasPrevious()	如果存在前一个元素,则返回 true;否则返回 false

续表

方　法	描　述
int nextIndex()	返回下一个元素的下标，如果不存在下一个元素，则返回列表的大小
Object previous()	返回前一个元素，如果前一个元素不存在，则引发一个 NoSuchElementException 异常
int previousIndex()	返回前一个元素的下标，如果前一个元素不存在，则返回 -1
void set(Object obj)	将 obj 赋给当前元素，这是上一次调用 next()方法或 previous()方法最后返回的元素

在通过迭代接口访问类集合之前，必须得到一个迭代对象。每一个 Collection 类都提供一个 iterator()方法，该方法返回一个类集合的迭代器。通过使用这个迭代器，可以访问类集合中的每一个元素。

使用迭代器循环遍历类集合的内容，步骤如下：

（1）通过调用类集合的 iterator()方法获得类集合的迭代器。

（2）建立一个含有 hasNext()方法的循环，只要 hasNext()返回 true，就进行循环迭代。

（3）在循环内部，通过调用 next()方法来得到每一个元素。

对于实现 List 接口的类集合，也可以通过调用 ListIterator 来获得迭代接口，列表迭代接口提供了前向或后向访问类集合的能力，并且可以修改元素。

【例 6.20】 Iterator 接口的使用。

```
package code0607;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ListIterator;
import java.util.Random;
public class IteratorDemo
{
    public static void main(String[] args)
    {
        ArrayList al=new ArrayList();
        Random r=new Random();
        for (int i=0; i<10; i++) {
            char c=(char) ('A'+r.nextInt(25));
            al.add(c);
        }
        System.out.print("原列表内容：");
        Iterator itr=al.iterator();
        while(itr.hasNext())
        {
            Object element=itr.next();
            System.out.print(element+" ");
        }
        System.out.println();
```

```
            ListIterator litr=al.listIterator();
            while(litr.hasNext())
            {
                Object element=litr.next();
                //把当前元素设置为小写
                litr.set(element.toString().toLowerCase());
            }
            System.out.print("修改后前向遍历列表:");
            itr=al.iterator();
            while(itr.hasNext())
            {
                Object element=itr.next();
                System.out.print(element+" ");
            }
            System.out.println();
            System.out.print("修改后后向遍历列表:");
            while(litr.hasPrevious())
            {
                Object element=litr.previous();
                System.out.print(element+" ");
            }
            System.out.println();
        }
    }
```

程序运行结果:

```
原列表内容:E N N U H Y D A S B
修改后前向遍历列表:e n n u h y d a s b
修改后后向遍历列表:b s a d y h u n n e
```

分析:在列表被修改之后,litr 指向列表的末端(当到达列表末端时,litr. hasNext()方法返回 false)。为了以反向遍历列表,程序继续使用 litr,但这一次,程序使用 hasPrevious()检测它是否有前一个元素。只要它有前一个元素,该元素就被获得并被显示出来。

6.7.5 映射接口

映射(map)是一个存储关键字/值对的集合。给定一个关键字,就可以得到它的值。关键字和值都是对象,每一对关键字/值称为一项。关键字必须是唯一的,但值是可以重复的。有些映射可以接收 null 关键字和 null 值,而有些映射则不行。映射接口如表 6-9 所示。

表 6-9 映射接口

接口	描述
Map	映射关键字到值
Map. Entry	描述映射中的项(关键字/值对),这是 Map 的一个内部接口
SortedMap	继承 Map 以便关键字按升序存储

1. Map 接口

Map 接口映射关键字到值。关键字(key)是以后用于检索值的对象。给定一个关键字和一个值,可以存储这个值到一个 Map 对象中。当这个值被存储以后,就可以使用它的关键字来检索它。由 Map 定义的主要方法如表 6-10 所示,当调用的映射中没有项存在时,其中的几种方法会引发一个 NoSuchElementException 异常。而当对象与映射中的元素不兼容时,引发一个 ClassCastException 异常。如果试图使用映射不允许使用的 null 对象时,则引发一个 NullPointerException 异常。

表 6-10 由 Map 定义的主要方法

方　　法	描　　述
void clear()	从映射中删除所有的关键字/值对
boolean containsKey(Object k)	如果映射中包含了关键字 k,则返回 true;否则返回 false
boolean containsValue(Object v)	如果映射中包含了值 v,则返回 true;否则返回 false
Set entrySet()	返回映射中的项的集合(Set),为映射提供了一个集合"视图"
Boolean equals(Object obj)	如果 obj 是一个 Map 并包含相同的项,则返回 true;否则返回 false
Object get(Object k)	返回与关键字 k 相关联的值
boolean isEmpty()	如果映射是空的,则返回 true;否则返回 false
Set keySet()	返回一个映射中关键字的集合(Set)
Object put(Object k, Object v)	将一个关键字/值对加入调用映射,覆盖原先与该关键字相关联的值
void putAll(Map m)	将所有来自 m 的项加入映射
Object remove(Object k)	删除关键字等于 k 的项
int size()	返回映射中项的个数
Collection values()	返回一个包含了映射中的值的类集合。这个方法为映射中的值提供了一个类集合"视图"

映射经常使用两个基本操作:get()和 put()。使用 put()方法可以将一个指定了关键字和值的项加入映射。为了得到值,可以通过将关键字作为参数来调用 get()方法返回该值。

映射不是集合,但可以获得映射的集合"视图"。为了实现这种功能,可以使用 entrySet()方法,它返回一个包含了映射中项的集合(Set)。为了得到关键字的"视图",可以使用 keySet()方法。为了得到值的"视图",可以使用 values()方法。类集合"视图"是将映射集成到集合框架的途径。

2. SortedMap 接口

SortedMap 接口继承了 Map,用于确保项按关键字升序排序。由 SortedMap 定义的方法如表 6-11 所示。

表 6-11 SortedMap 中定义的方法

方　　法	描　　述
Object firstKey()	返回映射的第一个关键字
SortedMap headMap(Object end)	返回一个映射,该映射包含了那些关键字小于 end 的映射项
SortedMap subMap(Object start, Object end)	返回一个映射,该映射包含了那些关键字大于等于 start 同时小于 end 的项
SortedMap tailMap(Object start)	返回一个映射,该映射包含了那些关键字大于等于 start 的项

排序映射允许对子映射(换句话说,就是映射的子集)进行高效的处理。使用 headMap()、tailMap()或 subMap()方法可以获得子映射。调用 firstKey()方法可以获得集合的第一个关键字,而调用 lastKey()方法可以获得集合的最后一个关键字。

6.7.6 实现 Map 接口的类

有几个类提供了映射接口的实现。可以被用做映射的类如图 6-3 所示,其中用斜体表示抽象类,一般不能直接使用,可以直接使用的类是 HashMap 和 TreeMap 等。

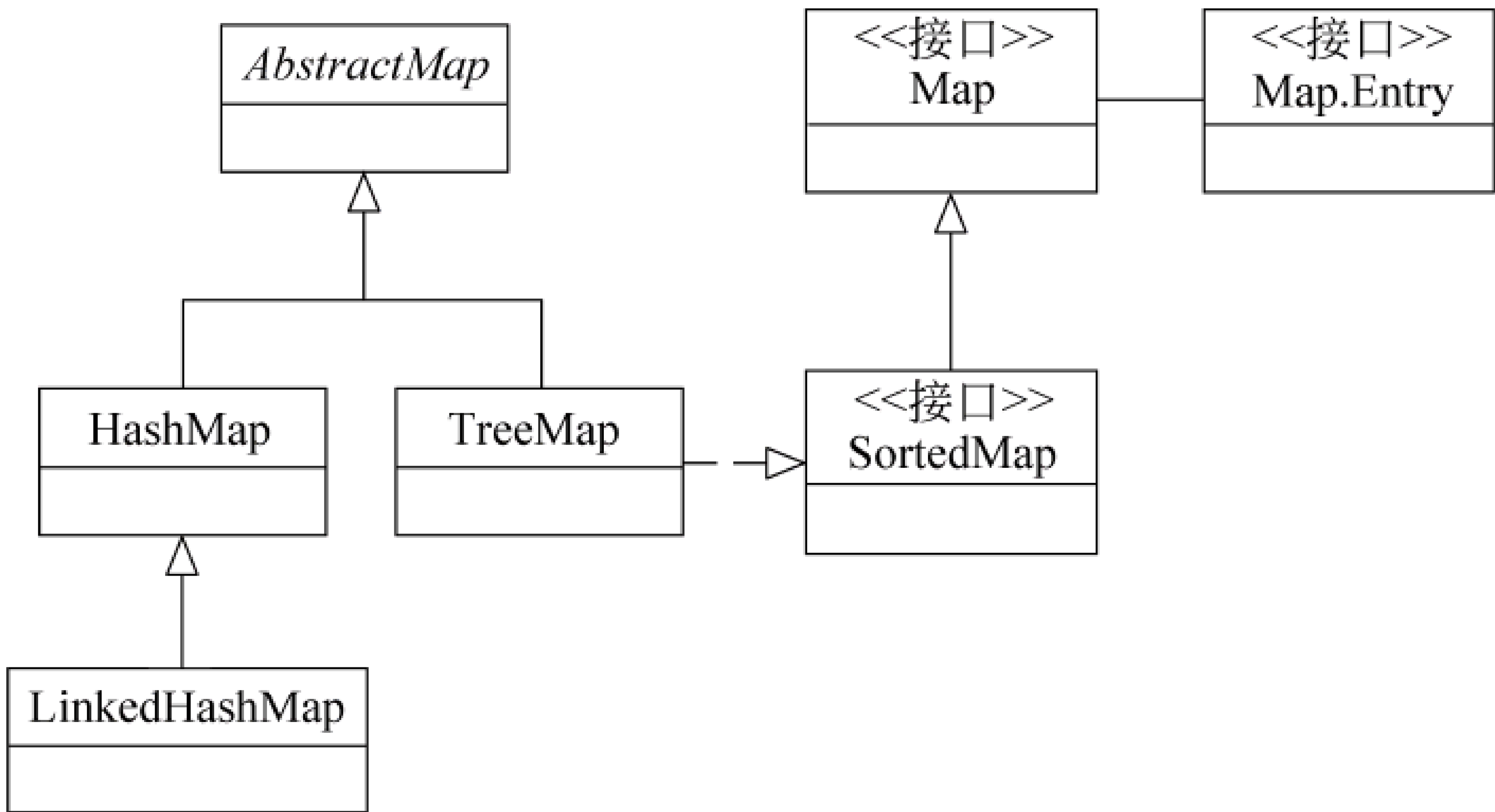


图 6-3 实现 Map 接口的类

1. HashMap 类

HashMap 类使用散列表实现 Map 接口,允许一些基本操作如 get()和 put()的运行时间保持恒定,即便对大型集合,也是这样的。

HashMap 实现 Map 接口,同时继承自 AbstractMap 类,它本身并没有增加任何新的方法。

注意:散列映射并不保证它的元素的顺序。因此,元素加入散列映射的顺序并不一定是它们被迭代函数读出的顺序。

【例 6.21】 HashMap 的使用。

```
package code0607;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Random;
```

```
import java.util.Set;
public class HashMapDemo
{
    public static void main(String[] args)
    {
        HashMap hm= new HashMap();
        Random r= new Random();
        //把元素加入映射
        hm.put("John", r.nextInt(10000));
        hm.put("Tom", r.nextInt(10000));
        hm.put("Jane", r.nextInt(10000));
        hm.put("Hall",r.nextInt(10000));
        hm.put("Smith", r.nextInt(10000));
        //得到映射项的集合
        Set set= hm.entrySet();
        Iterator i= set.iterator();
        //显示元素
        while(i.hasNext())
        {
            Map.Entry me= (Map.Entry)i.next();
            System.out.print(me.getKey()+ ": ");
            System.out.println(me.getValue());
        }
        //存 100 元到 John 的账户
        int balance= ((Integer)hm.get("John")).intValue();
        hm.put("John", new Integer((int) (balance+ 100)));
        System.out.println("John's new balance: "+ hm.get("John"));
    }
}
```

程序运行结果：

```
Tom: 8235
Smith: 5606
John: 9137
Hall: 5207
Jane: 747
John's new balance: 9237
```

分析：程序开始创建一个散列映射，然后将名字到账户值的映射增加到 HashMap 中。接下来，映射的内容通过使用函数 entrySet()而获得的项集合“视图”而显示出来，关键字和值通过调用由 Map.Entry 定义的 getKey()和 getValue()方法而显示，put()方法自动用新值替换与指定关键字相关联的原先的值。

2. TreeMap 类

TreeMap 类使用树实现 Map 接口。TreeMap 提供了按顺序存储关键字/值对的有效手段，同时允许快速检索。

注意：不像散列映射，树映射保证它的元素按照关键字升序排序。TreeMap 实现 SortedMap，而它本身并没有另外定义其他方法。

【例 6.22】 TreeMap 的使用。

```
package code0607;
import java.util.Iterator;
import java.util.Map;
import java.util.Random;
import java.util.Set;
import java.util.TreeMap;
public class TreeMapDemo
{
    public static void main(String[] args)
    {
        TreeMap   tm= new TreeMap();
        Random r= new Random();
        int f= 10000;
        //放入元素
        tm.put("E", f * r.nextDouble());
        tm.put("A", f * r.nextDouble());
        tm.put("B", f * r.nextDouble());
        tm.put("C", f * r.nextDouble());
        tm.put("F", f * r.nextDouble());
        tm.put("D", f * r.nextDouble());
        //得到关键字列表
        Set set= tm.keySet();
        //得到迭代器
        Iterator i= set.iterator();
        //通过迭代器显示 TreeMap 中的值
        while(i.hasNext())
        {
            String key= (String)i.next();
            System.out.println(key+ ": "+ ((Double)tm.get(key)).doubleValue());
        }
        //存 1000 到 D's 账户
        double balance= ((Double)tm.get("D")).doubleValue();
        tm.put("D", new Double(balance+ 1000));
        System.out.println("D's new balance: "+ tm.get("D"));
    }
}
```

程序运行结果：

```
A: 7050.924085349576
B: 5538.985344721475
C: 2662.3959343288884
```

```
D: 6880.620541129902
E: 4733.131385855935
F: 1839.547339561103
D's new balance: 7880.620541129902
```

提示：TreeMap 对关键字进行了排序。在使用 Iterator 访问关键字的集合时可以按关键字的自然顺序访问。

6.8 标准类实训任务

【任务描述】

设一个学生的信息包括学号、姓名、出生日期和性别等。把 n 个学生的信息放入一个集合中，可以根据学号对学生信息进行检索，并且可以根据出生日期对学生进行排序输出。

【任务分析】

对于每个学生的个人信息，可以定义一个类来表示，命名为 Student，其每个对象表示一个学生。题目要求按照学号对学生进行检索，这样可以以学号为键值，把学生的信息保存到一个 Map 中，出生日期可以用 Date 类表示。

为了完成检索和排序功能，定义一个工具类 StuManager，在其中定义如下几个函数：

(1) 初始化学生信息函数。

(2) 检索学生信息函数。

(3) 按照出生日期对学生进行排序函数。

【任务解决】

完整程序代码如下：

```
package code0608;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.List;
import java.util.TreeMap;
//定义学生类
class Student {
    int id;                     //学号
    String name;                //姓名
    String gender;              //性别
    Date birth;                 //出生日期
    public Student(int id, String name, String gender, Date birth) {
        this.id= id;
        this.name= name;
        this.gender= gender;
        this.birth= birth;
    }
    public String toString() {
```

```
            return id+ "- "+ name+ "- "+ gender+ "- "+ birth;
        }
    }

    //定义学生管理类
    public class StuManager {
        int STU_NUM= 10;
        TreeMap stuList= new TreeMap();
        //初始化学生信息
        void initStudents() {
            for (int i= 0; i< STU_NUM; i++) {
                int id= i;
                String name= "name"+ (int) (Math.random() * 1000);
                String gender= "女";
                if (i%2==0) {
                    gender= "男";
                }
                //随机生成出生日期
                Date birth= new Date(new Date().getTime()
                        + (long) (Math.random() * 10000000000001));
                Student stu= new Student(id, name, gender, birth);
                stuList.put(id, stu);
            }
        }
        Student searchByID(int id) {
            Student stu= (Student) stuList.get(id);
            return stu;
        }

        //按照日期进行排序,使用最简单的冒泡法
        List sortStudentByDate() {
            List stus= new ArrayList();
            //把学生加入列表
            stus.addAll(stuList.values());
            //使用冒泡法对列表中的学生排序
            for (int i= 0; i< stus.size(); i++) {
                for (int j= 0; j< stus.size(); j++) {
                    Student tmp1= (Student) stus.get(i);
                    Student tmp2= (Student) stus.get(j);
                    //比较出生日期
                    if (tmp1.birth.before(tmp2.birth)) {
                        stus.set(i, tmp2);
                        stus.set(j, tmp1);
                    }
                }
```

```
        }
        return stus;
    }

    public static void main(String args[]) {
        StuManager manager=new StuManager();
        manager.initStudents();
        //随机生成一个学号进行检索
        int id= (int) (Math.random() * 10);
        System.out.println("检索学号: "+id);
        Student stu=manager.searchByID(id);
        if (stu==null) {
            System.out.println("未检索到学生信息");
        } else
            System.out.println("检索到的学生信息是:"+stu);
        System.out.println("按照日期排序后的结果: ");
        List l=manager.sortStudentByDate();
        //遍历列表进行输出
        for (Iterator iterator=l.iterator(); iterator.hasNext();) {
            Student tmp= (Student) iterator.next();
            System.out.println(tmp);
        }
    }
}
```

程序结果:

```
检索学号: 1
检索到的学生信息是:1-name715-女-Wed Dec 30 11:39:13 CST 2026
按照日期排序后的结果:
2-name244-男-Mon Oct 15 20:12:02 CST 2012
4-name288-男-Thu Jun 23 03:20:27 CST 2016
0-name854-男-Fri Jun 10 01:36:03 CST 2022
9-name765-女-Wed Sep 14 06:06:37 CST 2022
8-name402-男-Thu Oct 30 00:59:14 CST 2025
1-name715-女-Wed Dec 30 11:39:13 CST 2026
3-name12-女-Fri Nov 17 01:23:58 CST 2028
5-name21-女-Wed Dec 21 09:28:15 CST 2033
7-name851-女-Fri Aug 24 05:48:20 CST 2040
6-name782-男-Wed May 15 18:53:45 CST 2041
```

习题与思考

1. 将一个字符串中的小写字母变成大写字母,并将大写字母变成小写字母。
2. 找出两个字符串中所有共同的字符。

3. 定义一个正则表达式识别字符串的邮件地址。

4. 编写一个程序,用 Map 实现学生成绩单的存储和查询,并且对成绩进行排序后存储到 TreeSet 中,求出平均成绩、最大值和最小值。

5. 给定一个整数－1234567,输出它的二进制、八进制和十六进制表示形式。

6. 编写一个程序,检查当前系统的 Java 版本和类路径。

7. 编写 Java 程序,在其中调用外部程序 cmd,并显示其输出结果。

8. 请使用 java. text. SimpleDateFormat 类对日期进行格式化,形式如 2005 年 8 月 10 日。

9. 编写一个日历程序,如图 6-4 所示。

2016 年 8 月

日	一	二	三	四	五	六
**	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

图 6-4　日历

第二篇　提 高 篇

第 7 章 异常处理

本章学习目标

在程序运行过程中,经常会出现由于硬件设备问题、软件设计错误、用户输入错误等原因引起的程序运行错误。为了能够处理这些问题,Java 引入了异常处理机制。本章主要讲述如何利用异常处理机制实现对 Java 应用中各种类型异常的发现、捕获和有效处理。通过本章的学习,应该重点掌握以下主要内容:

(1) 异常处理机制。

(2) 异常类及其层次关系。

(3) 异常的捕获和处理。

(4) 自定义异常类。

7.1 为什么需要异常处理

【例 7.1】 在盘点库存的盒子时,需要将盒子数量输入到程序中以计算盒子的总价值,编写这样一个盘点程序。

```
package code0701;

import java.io.* ;
import java.text.NumberFormat;

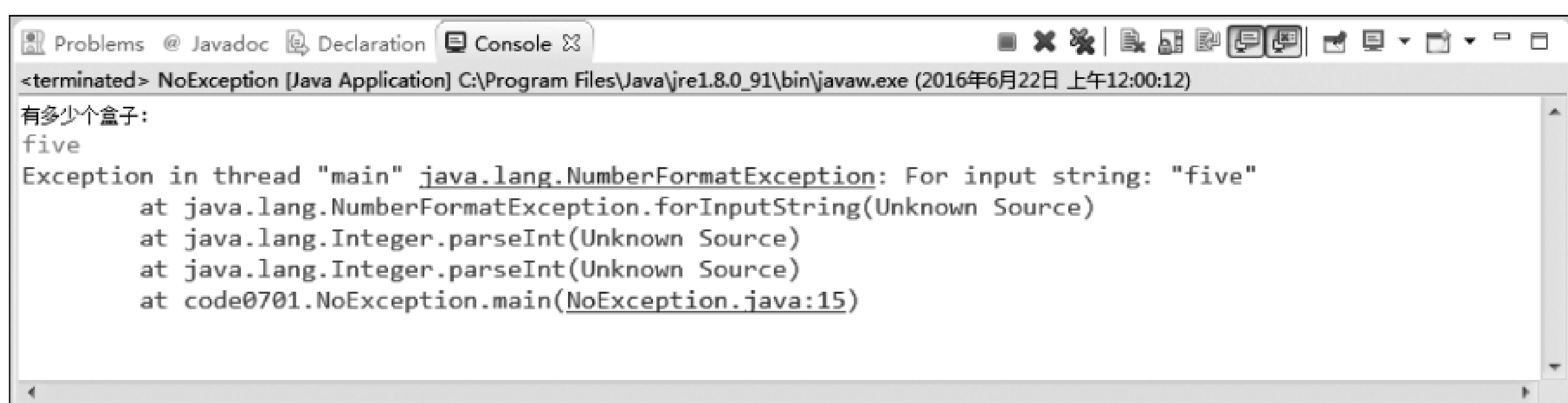
public class NoException {
    public static void main(String[] args) throws IOException {
        String numBoxesIn;
        int numBoxes;
        double boxPrice=3.25;
        NumberFormat currency=NumberFormat.getCurrencyInstance();
        System.out.println("有多少个盒子：");
        BufferedReader in=new BufferedReader(new InputStreamReader(System.in));
        //构建输入流以接收用户的输入数据
```

```
            numBoxesIn= in.readLine();
            numBoxes= Integer.parseInt(numBoxesIn);
            System.out.println("盒子的总价值是:");
            System.out.println(currency.format(numBoxes * boxPrice));
        }
    }
```

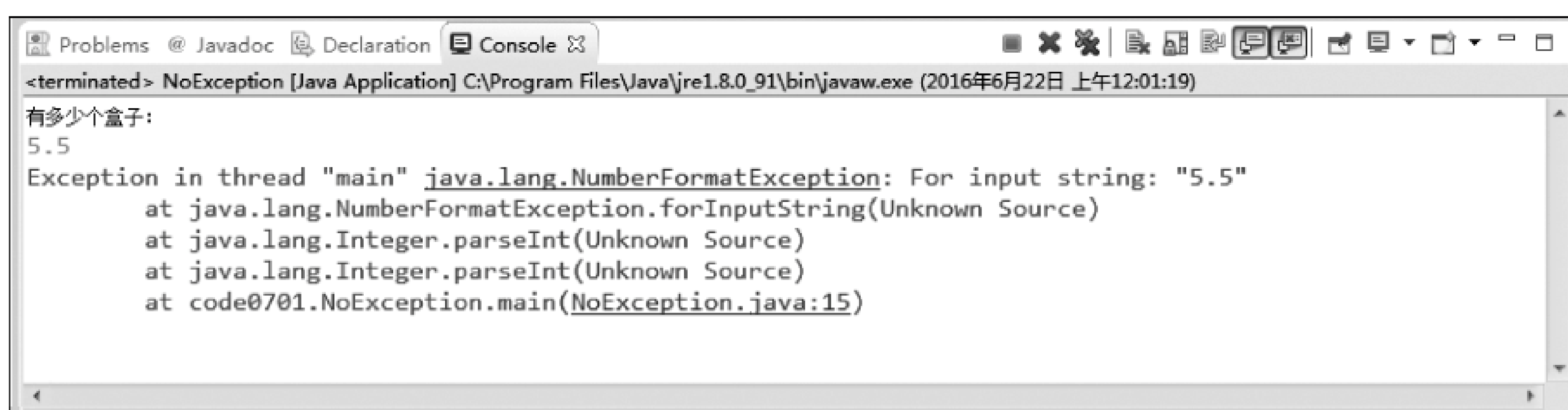
程序可能的运行结果如图 7-1 所示。



(a)



(b)



(c)

图 7-1　例 7.1 程序可能的运行结果

分析:该程序有一个缺陷,即当用户输入一个整数值时,一切正常;但当用户输入其他值时,程序就会彻底崩溃。为了在出现程序运行错误时,使其他所有代码能够继续完成其正常的工作,或者使程序能够体面地退出,最好能够对出现的错误进行处理,这时就需要引入异常处理。

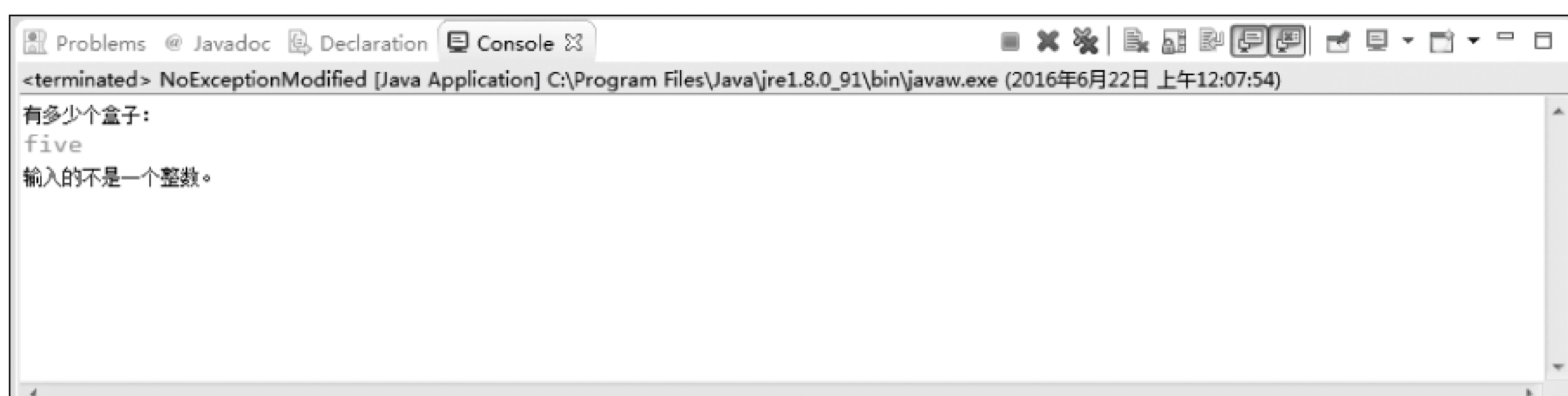
【例 7.2】　修改后的盘点程序。

```
package code0701;
import java.io.* ;
import java.text.NumberFormat;

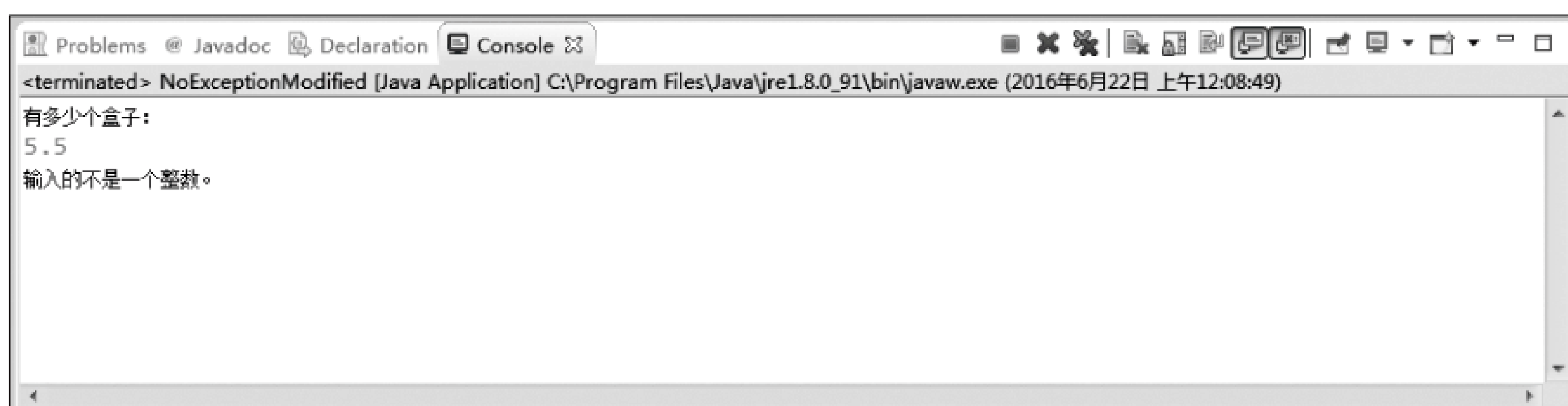
public class NoExceptionModified {
    public static void main(String[] args) throws IOException {
        String numBoxesIn;
```

```
        int numBoxes;
        double boxPrice= 3.25;
        NumberFormat currency= NumberFormat.getCurrencyInstance();
        System.out.println("有多少个盒子:");
        BufferedReader in= new BufferedReader(new InputStreamReader(System.in));
        numBoxesIn= in.readLine();
        try {
            numBoxes= Integer.parseInt(numBoxesIn);
            System.out.println("盒子的总价值是:");
            System.out.println(currency.format(numBoxes * boxPrice));
        } catch (NumberFormatException e) {
            System.out.println("输入的不是一个整数。");
        }
    }
}
```

程序可能的运行结果如图 7-2 所示。



(a)



(b)

图 7-2　例 7.2 程序可能的运行结果

分析:当用户输入错误时,程序将显示的提示是“输入的不是一个整数”。上述程序能够处理程序运行错误的关键是把 Integer.parseInt()方法调用封装在 try 语句块中。在执行 Integer.parseInt()方法时,计算机将监视异常。如果抛出了一个异常,那么计算机就从 try 语句块中跳到下面的 catch (NumberFormatException e)语句块并执行该语句块中的 println 语句,然后继续执行 catch 语句块后面的语句。

修改后的程序涉及如下 3 个关键问题:

(1) 在什么情况下会产生异常,参见 7.2 节。

(2) 如何使用 try…catch 处理异常,参见 7.3.3 节。

(3) 需要处理的异常类有哪些,参见 7.3.2 节。

此外,程序中还涉及从标准输入流中如何读取数据(System. in、InputStreamReader、BufferedReader)的问题,相关内容将在第 8 章中进行介绍。

7.2 异常概述

7.2.1 什么是异常

通常所说的异常(exception)指的是异常事件(exception event)。在程序运行时,很多情况都将导致异常事件的发生,例如:

(1) 想打开的文件不存在。

(2) 网络连接中断。

(3) 操作数超出预定范围。

(4) 访问的数据库打不开。

(5) 正在装载的类文件丢失。

可见,在程序中出现异常是普遍存在的。在 Java 语言中,对异常的处理有非常完备的机制,如下面的示例。

【例 7.3】 文件操作将产生异常。

```
package code0702;

import java.io.FileInputStream;

public class FileException {
    public static void main(String[] args) {
        FileInputStream fis= new FileInputStream("test.txt");
        int b;
        while((b= fis.read())!= - 1){
            System.out.print(b);
        }
        fis.close();
    }
}
```

当编译这个程序时,将出现错误信息,如图 7-3 所示。

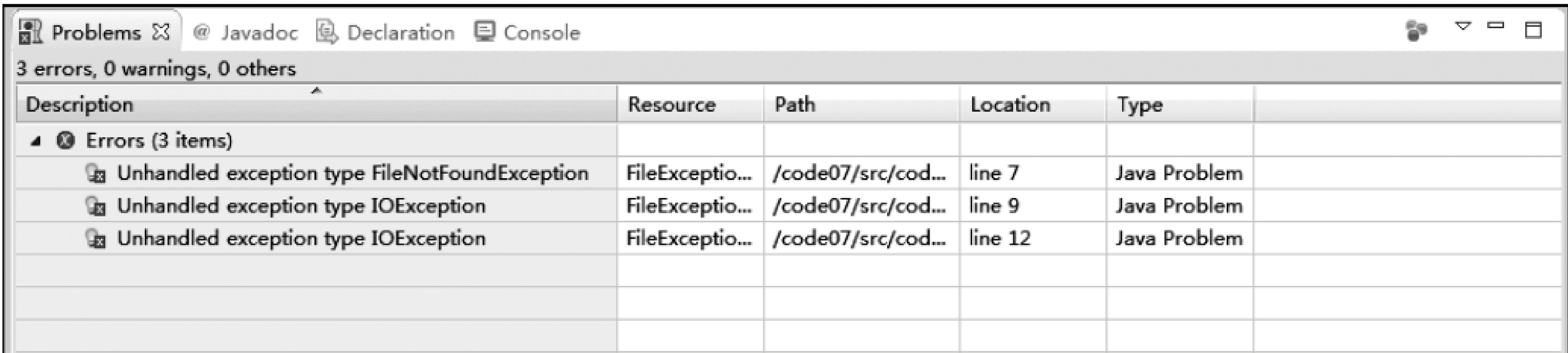


图 7-3 例 7.3 异常未处理的错误提示

分析：上述程序中出现的 3 个编译错误，是由于文件 test.txt 不存在或无法找到引起的，因此导致异常 ava.io.FileNotFoundException 和 java.io.IOException 的出现。

【例 7.4】 数组下标越界引发异常。

```
package code0702;

public class ArrayIndexException {
    public static void main(String[] args) {
        String words[]={"I","love","Java"};
        for(int i=0; i<4; i++){
            System.out.println(words[i]);
        }
    }
}
```

程序运行结果如图 7.4 所示。



图 7-4　例 7.4 程序运行时数组越界异常

分析：上述程序编译可以通过，但在运行时出现了异常。数组 langs 的长度为 3，当循环语句执行第 4 次时，导致数组下标越界，引发异常，并使程序终止。

上面两个例子中都遇到了异常，屏幕上所显示的 java.io.IOException、java.io.FileNotFoundException、java.lang.ArrayIndexOutOfBoundsException 分别指明了异常的类型和异常所在的包。对于某些异常，必须在程序中对它进行处理，否则编译程序会指出错误(如例 7.3)；而对另外一些异常，在程序中可以不做处理，直接交给运行时系统来处理(如例 7.4)，7.3 节将详细介绍异常的种类及其处理方法。

7.2.2　异常处理带来的好处

异常处理比传统的错误管理技术更具优势。使用异常处理可以将错误处理代码与正常代码分离。在传统的编程技术中，错误的检测、报告和处理代码往往会使程序的处理逻辑变得复杂。

【例 7.5】 对于读取文件的 readFile()方法，其处理逻辑的伪代码如下：

```
readFile {
    打开文件;
    得到文件的长度;
    分配足够大小的内存空间;
    读取文件内容到内存;
```

```
    关闭文件;
}
```


上述处理逻辑虽然简单,但却忽略了对可能出现的错误的处理。这些可能出现的错误包括不能打开文件、无法得到文件的长度、不能分配足够大小的内存空间、读取文件内容失败和不能关闭文件等。

因此,需要增加错误的检测、报告和处理代码,分别以传统编程技术和异常处理技术编写读取文件的 readFile()方法的伪代码,并比较两者的不同。

1. 基于传统编程技术的伪码形式

```
errorCodeType readFile {
初始化 errorCode=0;
打开文件;
    if (文件正确打开) {
得到文件的长度;
        if (已得到文件的长度) {
分配足够大小的内存空间;
            if (内存分配成功) {
读取文件内容到内存;
                if (读取失败) {
                    errorCode=-1;
                }
            } else {
                errorCode=-2;
            }
        } else {
            errorCode=-3;
        }
关闭文件;
        if (文件没有正确关闭 && errorCode==0) {
            errorCode=-4;
        } else {
            errorCode=errorCode and -4;
        }
    } else {
        errorCode=-5;
    }
    return errorCode;
}
```

2. 使用异常处理技术之后的伪码形式

```
readFile {
      try {
            打开文件;
            得到文件的长度;
```

```
            分配足够大小的内存空间;
            读取文件内容到内存;
            关闭文件;
    } catch (fileOpenFailed) {                //文件打开失败
        doSomething;
    } catch (sizeDeterminationFailed) {       //读取文件长度失败
        doSomething;
    } catch (memoryAllocationFailed) {        //分配内存空间失败
        doSomething;
    } catch (readFailed) {                    //读取文件内容失败
        doSomething;
    } catch (fileCloseFailed) {               //关闭文件失败
        doSomething;
    }
}
```

分析：通过比较上述两段代码，可以看到，在采用传统编程技术编写的代码中存在非常多的错误检测、报告和处理代码，使得正常的处理逻辑变得模糊，不便于程序的维护和修改。而采用异常处理技术编写的代码则在不同的地方编写正常的处理逻辑代码和错误处理代码，虽然异常处理技术并没有简化程序员进行错误处理的工作量，但却可以帮助程序员将错误处理工作组织得更加有效。

7.3 异常处理机制

7.3.1 Java 的异常处理机制

当一个方法在运行时出现异常事件时，该方法就会创建一个对象，并将该对象抛给运行时系统。这个对象称为异常对象(exception object)。异常对象存储了与错误有关的信息，包括错误类型以及异常事件出现时应用程序的状态和调用过程。创建异常对象并抛给运行时系统的过程称为抛出异常(throwing an exception)。能够处理异常的语句块称为异常处理程序(exception handler)。

Java 语言提供了捕获异常和声明抛弃异常两种处理异常的机制。

1. 捕获异常

当方法抛出异常后，运行时系统将沿着方法的调用栈逐层回溯，去查找能处理该异常的异常处理程序，这一过程称为捕获异常(catching exception)。

在图 7-5 中形成了一个方法调用栈，也就是 main()方法调用 f1()方法，f1()方法调用 f2()方法，f2()方法调用 f3()方法。如果 f3()方法抛出了异常，则运行时系统首先查看 f2()方法是否能处理；此时 f2()方法不包含异常处理程序，因此运行时系统继续查看 f1()方法是否能处理；f1()方法包含异常处理程序，因此异常被 f1()方法捕获下来并进行处理，此过程如图 7-6 所示。这是一种积极的异常处理机制。如果运行时系统在遍历了调用栈中所有的方法后，都没有找到合适的异常处理程序，则运行时系统将终止，相应地，Java 程序也将退出。

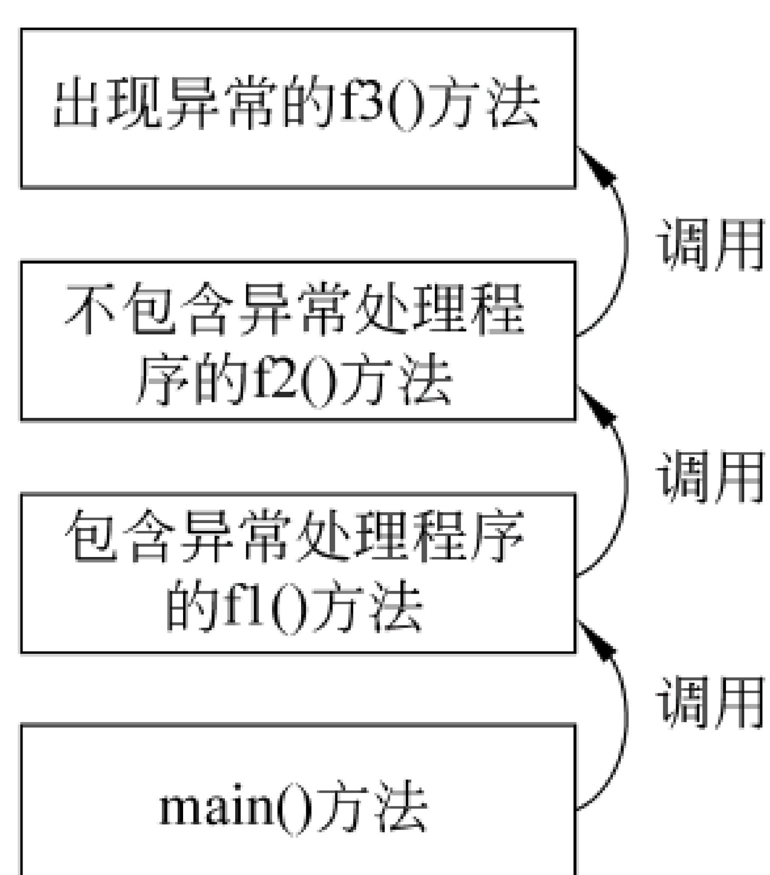


图 7-5 方法调用栈

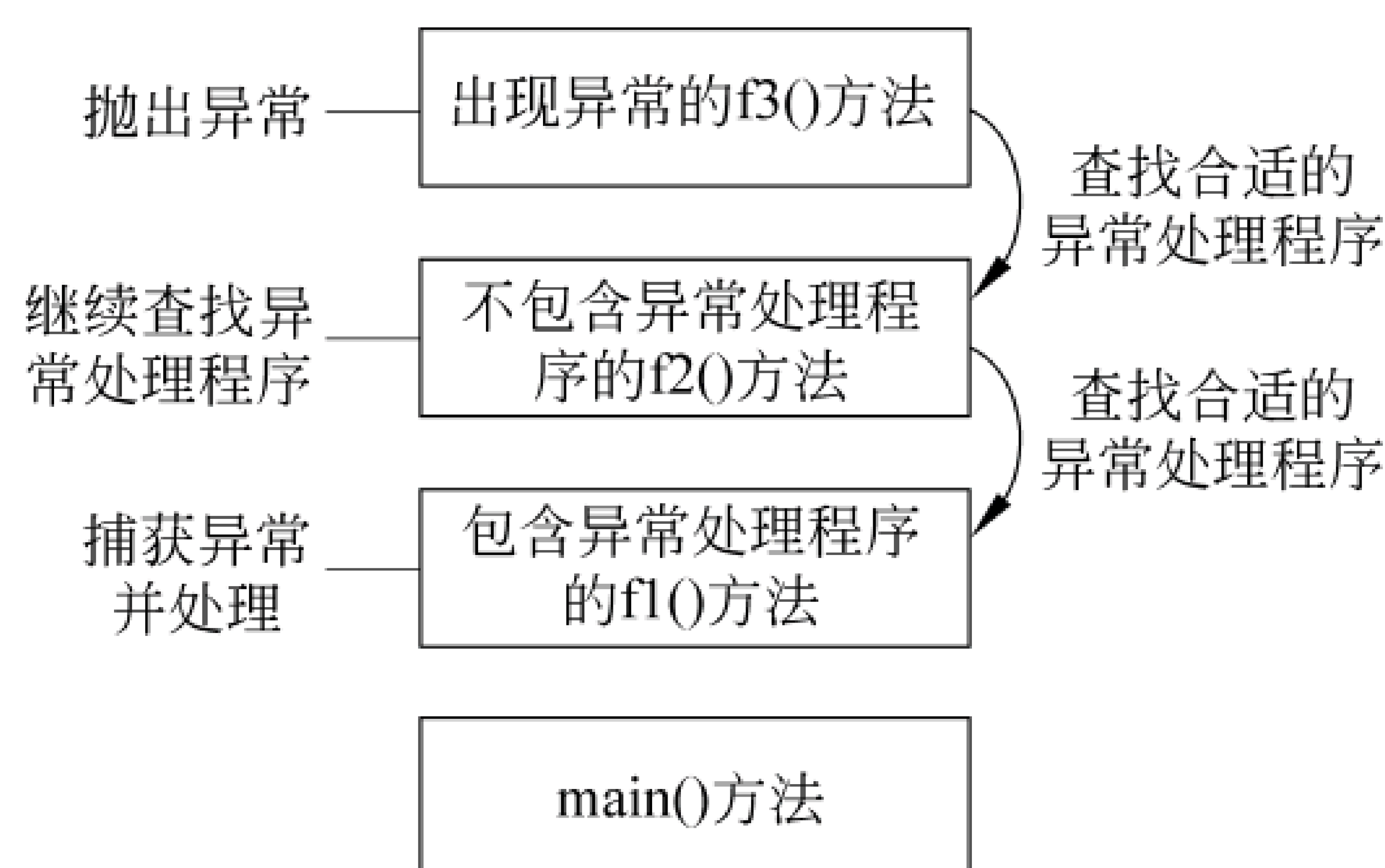


图 7-6 查找异常处理程序的过程

2. 声明抛弃异常

如果一个方法并不知道如何处理所出现的异常,则可以在声明该方法时声明抛弃该异常(specifying the exceptions thrown by a method)。

7.3.2 异常类的类层次

Java 是采用面向对象的方法来处理错误的,一个异常事件是由一个异常对象来代表的。在 Java 类库的每个包中都定义了自己的异常类,所有这些类都直接或者间接地继承于 Throwable 类,异常类的层次如图 7-7 所示。Java 中的异常可以分为 3 类:错误类(Error)、运行时异常类(Runtime Exception)和非运行时异常类(Checked Exception)。

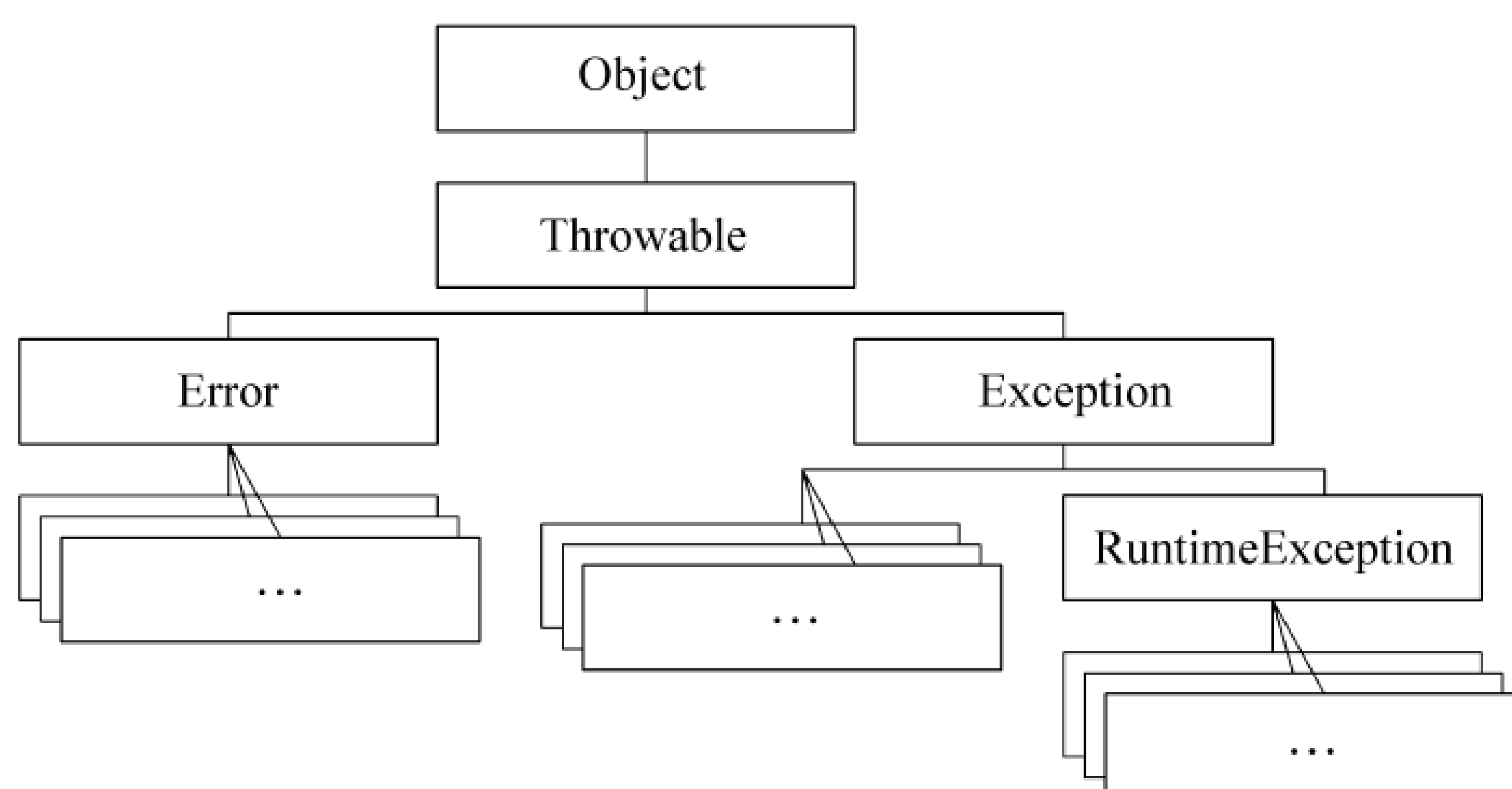


图 7-7 异常类的层次

1. 错误类

错误类继承于 Error 类。错误类往往是由程序外部的运行环境出现错误引起的,因此程序通常不必去处理这类异常。例如,一个应用程序成功地打开了一个输入文件,但因为硬件故障或者操作系统故障使得应用程序无法读取文件的内容,此时将产生 java.io.IOError。有时应用程序为了通知用户出现了软硬件故障,也会捕获该类异常。

常见的错误类有:AnnotationFormatError、AssertionError、AWTError、CoderMalfunctionError、FactoryConfigurationError、IOError、LinkageError、ServiceConfigurationError、ThreadDeath、TransformerFactoryConfigurationError 和 VirtualMachineError 等,用于标识动态链接失败、线程死锁、图形界面错误和虚拟机错误等。

2. 运行时异常类

运行时异常类继承自 RuntimeException 类，它代表了 Java 虚拟机在运行时所生成的异常。由于这类异常的出现很普遍，要求程序对这类异常全部做出处理可能对程序的可读性和高效性带来不好的影响，因此 Java 允许程序可以不对运行时异常做出处理。如例 7.4 中，程序对运行时异常 ArrayIndexOutOfBoundsException 并没有做出任何处理，而是直接交给运行时系统。当然在需要的时候，程序也可以处理运行时异常。

常见的运行时异常有：ArithmeticException（算术异常）、ClassCastException（类型转换异常）、ArrayIndexOutOfBoundsException（数组越界异常）、NegativeArraySizeException（指定数组维数为负值异常）和 NullPointerException（空指针异常）等。

3. 非运行时异常类

除了运行时异常以外的继承于 Exception 类的子类都统称为非运行时异常。对于这类异常，Java 编译器要求程序必须捕获或者声明抛弃异常。

常见的非运行时异常有：ClassNotFoundException（找不到类或接口所产生的异常）、IllegalAccessException（非法访问异常）和 IOException（输入/输出异常等）。

4. Throwable 类

Throwable 类是所有异常类的父类，常用的方法有以下几种。

String getMessage()：返回该 Throwable 对象详细信息。

void printStackTrace()：把该 Throwable 对象和它的跟踪情况打印到标准错误流。

void printStackTrace(PrintStream s)：把该 Throwable 对象和它的跟踪情况打印到指定的打印流。

void printStackTrace(PrintWriter s)：把该 Throwable 对象和它的跟踪情况打印到指定的打印流。

String toString()：返回该 Throwable 对象的简短字符串描述。

7.3.3 异常的处理

对于非运行时异常，程序必须捕获或者声明抛弃异常，否则程序无法通过编译。

1. 捕获异常

一个方法中如果对某种类型的异常对象提供了相应的处理代码，则这个方法可捕获该种异常。捕获异常是通过 try…catch…finally 语句实现的。其语法格式如下：

```
try{
…
}catch(exceptionType e1){
…
}catch(exceptionType e1){
…
}
…
finally{
…
}
```

1）try 语句块

捕获异常的第一步是用 try 语句块选定捕获异常的范围。在 try 语句块中的语句在执行的过程中可能会抛出异常。

2）catch 语句块

每个 try 语句块可以伴随一个或者多个 catch 语句块,用于处理 try 代码块中产生的异常事件。catch 语句只需一个形式参数来指明它所能够捕获的异常类型,这个类必须是 Throwable 的子类。运行时系统通过参数值把抛出的异常对象传递给 catch 语句块。

捕获异常的顺序与 catch 语句的顺序有关。当捕获到一个异常时,剩下的 catch 语句就不再进行匹配。因此,在安排 catch 语句的顺序时,首先应该捕获最特殊的异常,然后再捕获一般的异常。也就是说,一般先安排子类,再安排父类。例如,在捕获异常时如果写成如下形式:

```
}catch(IOException e){
    e.printStackTrace();
}catch(FileNotFoundException e){
    e.printStackTrace();
}
```

由于第一个 catch 语句首先得到匹配,第二个 catch 语句将不会被执行。编译时将会出现 Unreachable catch block 的错误提示信息。

3）finally 语句块

finally 语句块在 try 语句块退出后总会被执行,这保证了不论 try 语句块中是否有异常发生,finally 语句块都会被执行。finally 语句块在 try…catch…finally 结构中是可选的。finally 语句块最大的作用是防止资源泄露。在进行文件关闭或者其他资源回收操作时,一般都将相关代码放在 finally 语句块中,这样可以确保资源总是能够被正确回收,防止资源泄露。

此外,try…catch…finally 可以嵌套。

【例 7.6】 捕获异常的示例。

```
package code0703;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

public class CatchException {
    public static void main(String[] args) {
        FileInputStream fis=null;
        try {
            fis=new FileInputStream("test.txt");
            int b;
            while((b=fis.read()) !=-1) {
                System.out.print(b);
            }
        } catch (FileNotFoundException e) {
```

```
                System.out.println("Error: "+e.getMessage());
            } catch (IOException e) {
                System.out.println("Error: "+e.getMessage());
            } finally {
                try {
                    if (fis !=null) {
                        fis.close();
                    }
                } catch (IOException e) {
                }
            }
        }
    }
```

程序运行结果如图 7-8 所示。



图 7-8　例 7.6 程序运行结果

分析：由于需要打开的 test.txt 文件不存在，程序在创建文件输入流时会抛出 FileNotFoundException 的异常，该异常被 catch 语句块捕获并处理，打印出相应的错误描述信息。为了保证打开的文件能够被正确地关闭，将 fis.close()语句放在了 finally 语句块中。

2. 声明抛弃异常

如果在一个方法中发生了异常，但是该方法并不知道该如何处理该异常，此时该方法就应该声明抛弃异常，使得异常对象可以沿着方法的调用栈向后传播，直到有合适的方法捕获它为止。

声明抛弃异常是在一个方法声明的 throws 子句中指明的。例如：

```
public void writeList() throws IOException,ArrayIndexOutOfBoundsException {
...
}
```

throws 子句中可以同时指明多个异常，之间由逗号隔开。

【例 7.7】　声明抛弃异常的例子。

```
package code0703;

import java.io.FileInputStream;
import java.io.IOException;

public class ThrowException {
```

```
    public static void main(String[] args) throws IOException {
        FileInputStream fis= new FileInputStream("test.txt");
        int b;
        while((b= fis.read()) != - 1) {
            System.out.print(b);
        }
        fis.close();
    }
}
```

程序运行结果如图 7-9 所示。

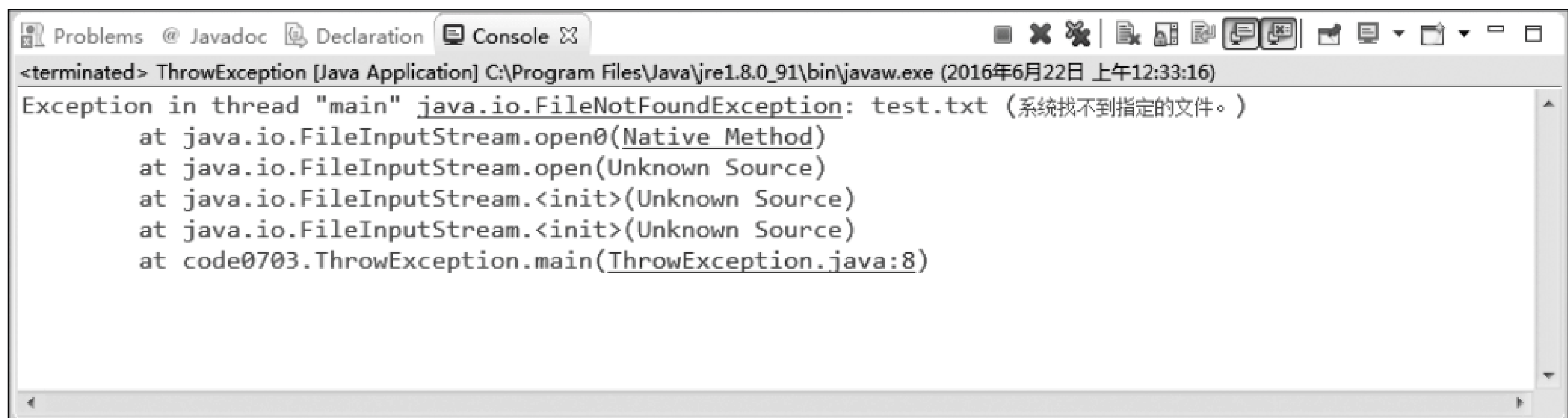


图 7-9　例 7.7 程序运行结果

分析:由于需要打开的“test. txt”文件不存在,程序在创建文件输入流时会抛出 FileNotFoundException 的异常,由于 main()方法采用了抛弃异常的处理,因此程序出现的异常就被提交给 Java 虚拟机处理。

3. 抛出异常

抛出异常就是产生异常的过程。首先要生成异常对象。异常对象或者由虚拟机生成,或者由某些类的实例生成,也可以在程序中生成。在方法中,抛出异常对象是通过 throw 语句来实现的。可以抛出的异常对象必须是 Throwable 类或者子类的实例。例如:

```
IOException e= new IOException();
throw e;
```

7.3.4　实用案例

【例 7.8】　找出数据文件中的最大值。

假定程序要从 data. txt 文件中读入一组整数,然后输出这组数据中的最大值。利用异常处理机制,对数据输入可能存在的异常情况进行处理,增强程序的鲁棒性。

```
package code0703;

import java.io.File;
import java.io.FileReader;
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.IOException;
```

```
public class FindMaxInteger {

    public static void main(String[] args) {
        String path= FindMaxInteger.class.getClassLoader().getResource("").getPath();
        File file= new File(path+ "data.txt");
        BufferedReader in= null;

        try {
            in= new BufferedReader(new FileReader(file));
            String line;
            int maxValue= 0;
            while((line= in.readLine())!= null)
            {
                int inputValue= Integer.parseInt(line);
                if(inputValue> maxValue)
                {
                    maxValue= inputValue;
                }
            }
            System.out.println("Max int is: "+ maxValue);
        } catch (FileNotFoundException e) {
            System.out.println("文件不存在。");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("输入输出错误。");
            e.printStackTrace();
        } catch (NumberFormatException e) {
            System.out.println("数字格式错误。");
            e.printStackTrace();
        }

    }

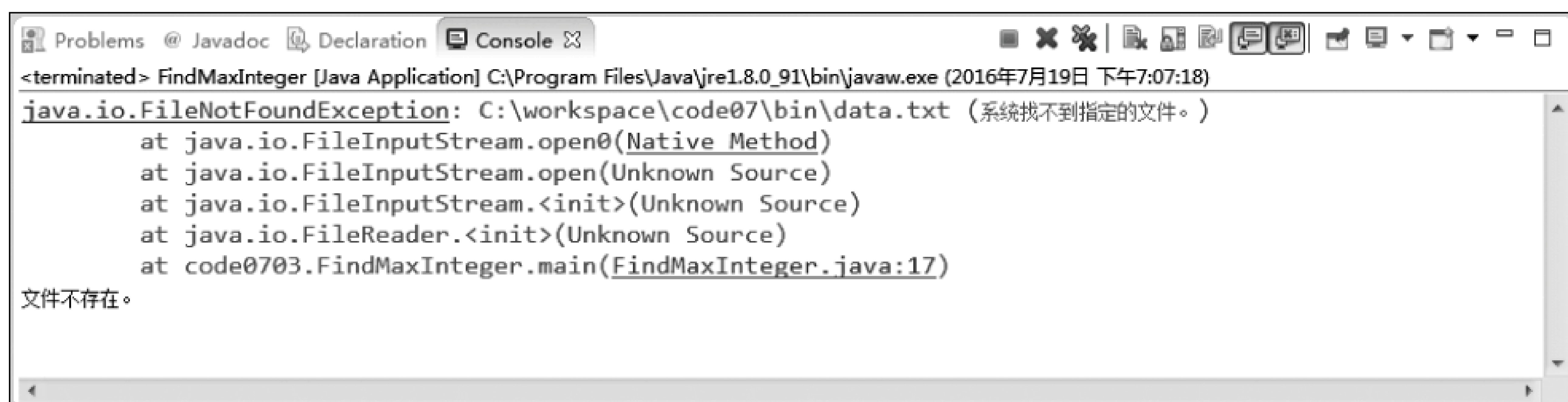
}
```

程序运行结果如图 7-10 所示。

分析：由于程序需要读取外部文件中的数据，对于指定的文件是否存在、文件中的数据是否合法等问题，编程时并不能保证这些问题不会出现。因此，通过异常处理机制对这些异常情况加以处理，在遇到异常情况时给用户适当的错误提示，从而可以提高程序的鲁棒性和人机交互的友好性。

7.4 自定义异常类

如果 Java 提供的系统异常类型不能满足程序设计的需求，那么可以设计自己的异常



(a)



(b)



(c)

图 7-10　例 7.8 可能的运行结果

类型。

从 Java 异常类的结构层次可以看出,Java 异常的公共父类是 Throwable。在程序运行中可能出现两种问题:一种问题是由硬件系统或者 Java 虚拟机导致的故障,Java 定义该故障为 Error,这类问题是用户程序不能解决的;另外一种问题是程序运行时错误,Java 定义为 Exception,这种情况下,可以通过程序设计的调整来实现异常处理。

因此,用户自定义的异常类型必须是 Throwable 的直接或者间接子类。Java 推荐用户的异常类型以 Exception 为直接父类。创建用户异常的方法如下:

```
class UserException extends Exception{
…      //语句
}
```

【例 7.9】 改写例 7.1 中的盘点程序,使之能够处理用户输入盒子数量为负数的错误。

```
import java.io.* ;
import java.text.NumberFormat;

public class Stocktaking {
    public static void main(String[] args) throws IOException {
```

```
        String numBoxesIn;
        int numBoxes;
        double boxPrice=3.25;
        NumberFormat currency=NumberFormat.getCurrencyInstance();

        System.out.println("有多少个盒子：");
        BufferedReader in=new BufferedReader(new InputStreamReader(System.
        in));                   //构建输入流以接收用户的输入数据
        numBoxesIn=in.readLine();
        try {
            numBoxes=Integer.parseInt(numBoxesIn);
            if (numBoxes<0) {
                throw new NegativeNumberException();
            }
            System.out.println("盒子的总价值是：");
            System.out.println(currency.format(numBoxes * boxPrice));
        } catch (NumberFormatException e) {
            System.out.println("输入的不是一个整数。");
        } catch (NegativeNumberException e) {
            System.out.println("盒子数量不可能为负数。");
        }
    }
}

class NegativeNumberException extends Exception {
}
```

程序的可能运行结果如图 7-11 所示。



图 7-11　例 7.9 程序可能运行结果

分析：通过引入用户自定义异常类 NegativeNumberException，对用户输入盒子数量为负数的错误情况进行了标识，使得程序可以利用异常处理对该错误进行处理。

7.5　异常处理实训任务

【任务描述】

编写一个应用程序，接收用户输入的一个正整数。如果用户输入的不是一个正整数，则

提示用户重新输入,直到输入正确为止。

【任务分析】

为了确保用户输入的是一个正整数,需要在程序中判断用户输入的类型,如果不是一个正整数,则抛出异常,由异常处理程序提示用户重新输入。因为需要不断地测试用户输入数据的类型,直到用户输入正整数为止,因此需要在程序中引入循环控制结构。

【任务解决】

完整示例代码如下:

```
package code0705;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class GetPositiveInteger {
    public static void main(String[] args) throws IOException {
        String inputStr;
        int inputNum;
        boolean gotGoodInput= false;
        do {
            try {
                System.out.println("请输入一个正整数:");
                BufferedReader in= new BufferedReader(new InputStreamReader(
                        System.in));
                inputStr= in.readLine();
                inputNum= Integer.parseInt(inputStr);
                if (inputNum< = 0) {
                    throw new NegativeIntegerException();
                }
                gotGoodInput= true;
            } catch (NumberFormatException e) {
                System.out.println("输入的不是一个整数,请重新输入。");
            } catch (NegativeIntegerException e) {
                System.out.println("输入的不是一个正整数,请重新输入。");
            }
        } while(! gotGoodInput);
    }
}

class NegativeIntegerException extends Exception {
}
```

程序运行结果如图 7-12 所示。

习题与思考

1. 什么是异常?简述 Java 的异常处理机制。

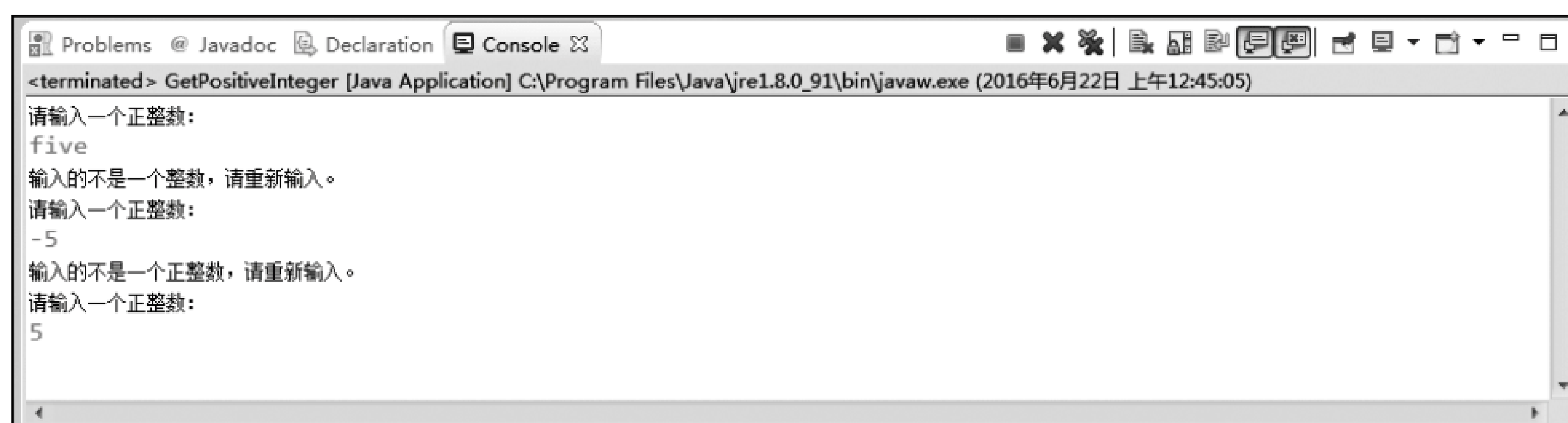


图 7-12　异常处理实训任务运行结果

2. 下面的代码是否正确？

```
try {

} finally {

}
```

3. 下面的 catch 语句可以捕获哪些类型的异常？如果这样使用，会有什么不足之处？

```
catch (Exception e) {

}
```

4. 修改下面的代码，使之能够通过编译。

```
public static void cat(File file) {
    RandomAccessFile input=null;
    String line=null;

    try {
        input=new RandomAccessFile(file, "r");
        while((line=input.readLine()) !=null) {
            System.out.println(line);
        }
        return;
    } finally {
        if (input !=null) {
            input.close();
        }
    }
}
```

第 8 章 输入输出处理

本章学习目标

绝大部分程序都离不开信息的输入和输出(I/O)。例如,从键盘上读取数据、在网络上交换数据、打印报表、读写文件信息等,都要涉及输入输出的处理。在 Java 中,输入输出信息的处理都是通过数据流来实现的。本章主要介绍 Java 数据流的基本概念以及 java.io 中的各种输入输出流的应用。通过本章的学习,应该重点掌握以下主要内容:

(1) 流的基本划分,字节流和字符流的不同应用场景。

(2) 文件输入输出字节流的应用。

(3) 标准输入输出流的应用。

(4) 文件字符流和 Scanner 的使用。

(5) 串行化。

8.1 流的作用

在前面的 Java 程序中,程序的运行结果一般输出到控制台,这是一种临时输出措施。有时需要把程序的运算结果长久保存,可以运用 Java 提供的输入输出流的功能把结果保存到文件中。

【例 8.1】 把程序运行结果保存到文件中。

分析: 如果把运行结果输出到控制台,可以使用 System.out,如果长久保存就应输出到文件中,使用文件输出流。

问题解决如下:

```
package code0801;
import java.io.FileWriter;
public class SimpleFileStream {
    public static void main(String[] args) throws Exception {
        int sum=0;
        for (int i=1; i<=100; i++) {
            sum+=i;
```

```
        }
        //输出控制台
        System.out.println("计算结果 1: "+ sum);
        //构建文件输出流,输出内容到文件 f1.txt 中
        FileWriter fout=new FileWriter("f1.txt");
        //执行输出操作
        fout.write("计算结果 2: "+ sum);
        fout.close();
    }
}
```

程序的运行结果有两部分,一部分输出到控制台如下:

```
计算结果 1: 5050
```

另一部分的输出结果在文件 f1.txt 中,打开文件 f1.txt 看到下面的结果:

```
计算结果 2: 5050
```

该问题的解决涉及下面几个关键点:

(1) 选择并创建合适的文件输出流对象,参见 8.4 节和 8.5 节。

(2) 使用输出流的输出方法,本例使用的是 write 方法,参见 8.4 节和 8.5 节。

(3) 关闭输出流,本例使用的是 close()方法,参见 8.4 节和 8.5 节。

8.2 流的划分

数据流(Stream)是一组有顺序的、有起点和终点的字节集合,是对输入和输出的总称和抽象。

一般地,数据流分为输入流(InputStream)和输出流(OutputStream)。输入流只能读,不能写,而输出流只能写不能读。

Java 程序通过流来连接设备完成输入和输出。尽管连接的物理设备不尽相同,但所有流均具备相同的操作方式。一个输入流能够抽象多种不同类型的输入:磁盘文件、键盘或网络套接字。同样,一个输出流可以输出到控制台、磁盘文件或相连的网络。

Java 主要定义了两种类型的流:字节流和字符流。字节流以字节为基本处理单位。例如,使用字节流读取或输出二进制数据。字符流以字符为基本处理单位,采用了统一的编码标准。在某些场合,字符流比字节流更有效。

提示: *字符流为处理字符提供方便有效的方法。字符流也可以由字节流转化而来。*

1. 字节流

字节流在顶层有两个抽象类:InputStream 和 OutputStream,定义了所有字节流的关键方法。最重要的两种方法是 read()和 write(),它们分别用于读和写字节。两种方法都在 InputStream 和 OutputStream 中被定义为抽象方法,它们被不同的子类重写,可应用于不同的场景。

常用的输入字节流的描述如表 8-1 所示。

表 8-1　输入字节流的描述

流	描　　述
InputStream	表示输入字节流的抽象类,是其他流的父类
FileInputStream	文件字节输入流
FilterInputStream	过滤字节输入流,提供扩展功能的输入流
BufferedInputStream	缓冲输入流,可以在其他流的基础上构建缓存,是 FilterInputStream 的子类
DataInputStream	包含读取 Java 标准数据类型方法的输入流,是 FilterInputStream 的子类
ObjectInputStream	对象输入流,用于对象串行化

输出字节流的描述如表 8-2 所示。

表 8-2　输出字节流的描述

流	描　　述
OutputStream	表示输出字节流的抽象类,是其他流的父类
FileOutputStream	文件字节输出流
FilterOutputStream	过滤字节输出流,提供扩展功能的输出流
BufferedOutputStream	缓冲输出流,在其他流的基础上构建缓冲功能,是 FilterOutputStream 的子类
DataOutputStream	包含 Java 标准数据类型方法的输出流,是 FilterOutputStream 的子类
PrintStream	打印流,包含 print()和 println()的输出流,是标准输出流 System.out 使用的流
ObjectOutputStream	对象输出流,用于对象串行化

2. 字符流

字符流顶层有两个抽象类:Reader 和 Writer,定义了所有字符流的关键方法。其中,两个最重要的方法是 read()和 write(),它们分别进行字符数据的读和写。这些方法被不同的子类重写,可应用于不同的场景。

字符输入流的描述如表 8-3 所示。

表 8-3　字符输入流的描述

流	描　　述
Reader	表示字符输入流的抽象类,是其他流的父类
BufferedReader	带缓冲功能的字符输入流
InputStreamReader	字节流向字符流转化的流
FileReader	文件字符输入流,是 InputStreamReader 的子类

字符输出流的描述如表 8-4 所示。

表 8-4 字符输出流的描述

流	描　述
Writer	描述字符输出流的抽象类,是其他流的父类
BufferedWriter	带缓冲功能的字符输出流
OutputStreamWriter	字符输出流向字节输出流的转换
PrintWriter	打印字符输出流,类似 PrintStream
FileWriter	文件字符输出流

由于涉及的输入输出流比较多,本章主要介绍文件输入输出字节流、标准输入输出流、文件字符流和 Sanner 类的使用。

8.3 标准输入输出流

Java 通过系统类 System 实现标准输入输出的功能,定义了 3 个流变量,in、out 和 err。这 3 个流在 Java 中都定义为静态变量,可以直接通过 System 类进行调用。System.in 表示标准输入,通常指键盘数据输入;System.out 表示标准输出,通常指把数据输出到控制台或者屏幕;System.err 表示标准错误输出,通常指把错误信息输出到控制台或者屏幕。

8.3.1 标准输入

System.in 作为字节输入流类 InputStream 的对象实现标准输入,通过 read()方法从键盘接收数据。

```
int read()
int read(byte b[])
int read(byte b[], int offset, int len)
```

【例 8.2】 从标准输入读取数据。

```
package code0803;
import java.io.IOException;
public class StdInput
{
    public static void main(String[] args) throws IOException
    {
        System.out.println("input:");
        byte b[]=new byte[512];
        int count=System.in.read(b);
        System.out.println("Output");
        for (int i=0;i<count;i++)
        {
            System.out.print(b[i]+" ");
        }
```

```
            System.out.println();
            for (int i=0;i< count;i++)
            {
                System.out.print((byte)b[i]+" ");
            }
            System.out.println("count="+count);
        }
    }
```

程序运行结果:

```
input:
abcd中
Output
97 98 99 100 -42 -48 13 10
97 98 99 100 -42 -48 13 10 count=8
```

分析:程序运行时,从键盘输入 5 个字符 a、b、c、d、中并按 Enter 键,保存在缓冲区 b 中的元素个数 count 为 8,Enter 占用两个字节,汉字占用两个字节。

8.3.2 Scanner 类封装标准输入流

有时用户需要从标准输入中读取一个字符、整数或浮点数等具体类型的数据,而 System.in 作为标准输入流,是一个 InputStream 类的对象,其 read()方法的主要功能是读取字节和字节数组,不能直接得到需要的数据(如整型、浮点型)。此时,需要另外一个类 java.util.Scanner 的配合。Scanner 类可以对标准输入流 System.in 的数据进行解析,得到需要的数据。

【例 8.3】 从标准输入读取一个整数和浮点数,并计算它们的乘积。

```
package code0803;
import java.util.Scanner;
public class ScannerSystemIn {
    public static void main(String[] args) {
        Scanner sin=new Scanner(System.in);      //封装标准输入流
        int a=sin.nextInt();                     //读取整数
        double b=sin.nextDouble();               //读取浮点数
        double result=a * b;
        System.out.println("a*b="+result);
    }
}
```

程序运行结果:

```
10
123.456
a*b=1234.56
```

分析:结果中的第一行和第二行是从控制台输入的数据,第三行是程序的输出结果。

今后从标准输入读取数据都可以按照这个例子的方式使用。Scanner 类还提供了很多获得其他类型数据的方法,例如 nextBoolean()、nextFloat()、nextByte()等。

8.3.3 标准输出

System.out 作为打印输出流 PrintStream 的对象实现标准输出,其中定义了 print 和 println 方法,支持 Java 任意基本类型作为参数。

```
public void print(int i);
public void println(int i);
```

两者的区别在于,println 在输出时加一个回车符,它已经在本书的前面章节多次使用过,也是标准输出的最常用方法。

另外,增加了 printf()方法,支持数据的格式化输出。

```
public PrintStream printf(String format, Object … args)
public PrintStream printf(Locale l, String format, Object … args)
```

该方法支持可变参数,即方法的参数个数是可变的。format 参数指名输出格式,args 是输出参数列表。

format 字符串的格式如下:

```
%[argument_index$][flags][width][.precision]conversion
```

其中,argument_index 用十进制整数表示参数在参数列表中的位置,第一个参数用%1$表示,第二个用%2$表示,以此类推,如果只有一个参数,可以只保留%;flags 是调整输出格式的字符集合;width 是一个非负整数,表示输出的最小字符数;precision 是一个非负整数,用于限制字符个数,如果输出浮点数,则是小数点后面的位数。conversion 是一个转换字符,表示参数如何被格式化。该方法在格式化输出时,使用了类 java.util.Formatter 的功能。另外,String 类的 format()函数和这里的 printf()使用相同的方法格式化字符串。

常用的转换符如下。

(1) d:十进制整数。

(2) x:十六进制整数。

(3) f:浮点数。

(4) s:字符串。

(5) c:字符。

(6) t:格式化日期,后面还可以跟其他日期转换符。

详细的转换字符(conversion)参见类 java.util.Formatter 的文档。

8.3.4 实用案例

【例 8.4】 数据的格式化输出。

```
package code0803;
import java.util.Date;
public class PrintfDemo {
```

```
    public static void main(String[] args) {
        double d1=23456789.567;

        int i=65;
        //用逗号作为分隔符,格式化浮点数、字符
        System.out.printf("%1$,.2f\n十六进制%2$x=%2$c\n", d1,i);
        Date c=new Date();
        //格式化日期
        System.out.printf("日期:%1$tF 时间:%1$tT %1$tA\n", c);
        String s=String.format("日期:%1$tF 时间:%1$tT %1$tA\n", c);
        System.out.println(s);
    }
}
```

程序运行如下:

```
23,456,789.57
十六进制 41=A
日期:2016-06-17 时间: 15:14:05 星期五
日期:2016-06-17 时间: 15:14:05 星期五
```

分析:格式化数字时:%表示此处有参数,逗号表示数字的千位分隔符,是一个 Flag;“.2”表示数据精度,f 表示格式化浮点数,x 表示十六进制整数;c 表示格式化字符,可以把整数输出为字符;格式化日期时:%1 表示第一个参数,t 表示格式化日期,F 表示输出日期格式如 2015-07-22,T 表示输出时间格式如 16:43:38,A 表示提取星期几。其他转换符的详细情况请参考类 java.util.Formatter 的文档。

8.4 字节流使用

字节流为处理字节式输入输出提供了丰富的环境,比较典型的是文件字节流的使用。文件字节流包含文件输入字节流 FileInputStream 和文件输出字节流 FileOutputStream。在介绍文件字节流之前,先了解一个表示文件对象的类 java.io.File,它虽然不能读取文件的内容,但是可以获取文件的信息并对文件进行操作。

8.4.1 File 类

在进行文件操作时,需要知道一些关于文件的信息。File 类提供了一些方法来操作文件和获取文件的信息。对于目录,Java 把它视为一种特殊类型的文件,即文件名单列表。

File 类直接处理文件和文件系统。通过 File 类的方法,可以得到文件或者目录的描述信息,包括名称、所在路径、可读写性和长度等,还可以创建目录、创建文件,改变文件名、删除文件和列出目录中的文件等。

1. 构造方法

下面的构造函数可以用来生成 File 对象:

```
File(String path)
```

```
File(String dir, String filename)
File(File dir, String filename)
File(URI uri)
```

这里,dir 是文件所在的目录,filename 是文件名,path 是文件的路径名。

下面的例子创建了 4 个 File 对象:f1、f2、f3 和 f4。第一个 File 对象是由仅有一个路径参数的构造函数生成的。第二个 File 对象有两个参数:目录和文件名。第三个 File 对象的参数包括指向 f1 文件的目录及文件名。f3 和 f2 指向相同的文件。第四个对象是用一个 uri 参数构造一个文件。

```
File f1=new File("D:/Java");
File f2=new File("D:/Java","test.txt");
File f3=new File(f1,"test.txt");
File f4=new File("file://D:/Java/test.txt");
```

注意: Java 能正确处理 UNIX 和 Windows/DOS 约定的路径分隔符。如果在 Windows 版本的 Java 下用斜线(/),路径处理依然正确。记住,如果在 Windows/DOS 使用反斜线(\),需要在字符串内使用它的转义序列(\\)。Java 约定用 UNIX 和 URL 风格的斜线作为路径分隔符。

File 定义了很多获取 File 对象标准属性的方法。例如,getName()返回文件名;getParent()返回父目录名;exists()在文件存在的情况下返回 true,反之返回 false。

2. File 类提供的方法

创建一个文件对象后,可以用 File 类提供的方法来获得文件相关信息,对文件进行操作。File 类的主要方法如表 8-5 所示。

表 8-5　File 类的主要方法

方　法	描　述
boolean canRead()	测试文件是否可读
boolean canWrite()	测试文件是否可写
boolean createNewFile()	创建文件
static File createTempFile(String prefix, String suffix)	创建临时文件
boolean delete()	删除文件
boolean equals(Object obj)	比较两个文件对象
boolean exists()	测试文件是否存在
File getAbsoluteFile()	返回绝对文件名
String getAbsolutePath()	返回绝对路径
String getName()	返回文件名(不包括路径)
String getParent()	返回父目录
String getPath()	返回路径

续表

方　　法	描　　述
boolean isAbsolute()	是否绝对路径
boolean isDirectory()	是否是目录
boolean isFile()	是否是文件
boolean isHidden()	是否隐藏文件
long lastModified()	上次修改时间,从 1970 年 1 月 1 日开始的标准时间(UTC)的毫秒数
long length()	文件长度
boolean renameTo(File dest)	重命名文件
String toString()	把文件对象转化成字符串

【例 8.5】 File 类的使用。

```
package code0804;
import java.io.File;
import java.io.IOException;
public class FileDemo {
    public static void main(String[] args) throws IOException {
        File dir=new File("src/code0804");
        File f1=new File(dir, "FileDemo.java");
        System.out.println(f1);
        System.out.println("exist: "+f1.exists());
        System.out.println("name:   "+f1.getName());
        System.out.println("path:   "+f1.getPath());
        System.out.println("abosolute path:"+f1.getAbsolutePath());
        System.out.println("parent:   "+f1.getParent());
        System.out.println("is a file:   "+f1.isFile());
        System.out.println("is a directory:       "+f1.isDirectory());
        System.out.println("length:    "+f1.length());
        File temp=File.createTempFile("临时文件", ".tmp");
        System.out.println("abosolute path:       "+temp.getAbsolutePath());
        System.out.println("length:   "+temp.length());
    }
}
```

程序运行结果:

```
src\code0804\FileDemo.java
exist: true
name:   FileDemo.java
path:   src\code0804\FileDemo.java
abosolute path:C:\Users\yangrl\eclipseworkspace\Ch08\src\code0804\FileDemo.java
parent: src\code0804
is a file: true
```

```
is a directory:      false
length:    904
abosolute path: C:\Users\yangrl\AppData\Local\Temp\临时文件 3468493380817714287.tmp
length:   0
```

3. 目录

File 对象也可以表示一个目录,目录是一个包含其他文件和路径列表的 File 类。当创建一个 File 对象且它是目录时,isDirectory() 方法返回 true。这种情况下,可以调用该对象的 list()方法来提取该目录内部其他文件和目录的列表。该方法有两种形式。

第一种形式如下:

```
String[] list()
```

文件列表通过一个 String 对象数组返回。

第二种形式如下:

```
File[] listFiles()
```

文件列表通过一个 File 对象的数组返回。

有时需要列出目录下指定类型的文件,例如.java、.class 等扩展名的文件。可以使用 File 类的下述 3 个方法,列出指定类型的文件:

```
String[] list(FilenameFilter FFObj)
File[] listFiles(FilenameFilter FFObj)
File[] listFiles(FileFilter FObj)
```

第一种方法用字符串形式返回目录下的指定类型的文件;第二种方法用 File 对象数组的形式返回目录下的指定类型的文件;第三种方法也是用 File 对象数组的形式返回目录下的指定类型的文件,和第二种方法不同的是,参数类型不同。

参数 FFObj 是一个实现 FilenameFilter 接口的类对象。

FilenameFilter 仅定义了一个方法,即 accept()。调用 list 方法列出文件时,将调用 accept()方法检查该文件名称是否符合 accept 方法指定的名字要求。它的通常形式如下:

```
boolean accept(File directory, String filename)
```

如果返回 true,名称为 filename 的文件包含在返回列表中,反之,不包含在返回列表中。

FFObj 是一个实现了 FileFilter 接口的类的对象。FileFilter 只定义了一个 accept()方法,该方法针对目录中的每个文件调用一次,它不仅可以根据文件名过滤文件,也可以根据文件的其他信息过滤文件。它的通常形式如下:

```
boolean accept(File path)
```

如果返回值是 true,path 代表的文件包含在返回列表中,反之,不包含在返回列表中。

【例 8.6】 列出目录中特定类型的文件。

```
package code0804;
import java.io.File;
import java.io.FileFilter;
```

```
import java.io.FilenameFilter;
public class DirDemo
{
    public static void main(String[] args)
    {
        File dir=new File("bin/code0803");
        System.out.println("列出目录"+dir+"中的 class 文件");
        Filter filter=new Filter("class");
        File fs1[]=dir.listFiles(filter);
        display(fs1);
        System.out.println("列出目录"+dir+"中的子目录");
        DirFilter filter2=new DirFilter();
        File fs2[]=dir.listFiles(filter2);
        display(fs2);
        //在创建一个子目录
        File f=new File(dir,"_newDir");
        System.out.println(f.mkdir());
        File fs3[]=dir.listFiles(filter2);
        display(fs3);
    }
    public static void display(File[] fs)
    {
        for (int i=0; i<fs.length; i++)
        {
            if (fs[i].isDirectory())
                System.out.println("目录:"+fs[i]);
            else
                System.out.println("文件:"+fs[i]);
        }
    }
}
//定义扩展名过滤器
class Filter implements FilenameFilter
{
    String extent;
    Filter(String extent)
    {
        this.extent=extent;
    }
    public boolean accept(File dir, String name)
    {
        return name.endsWith("."+extent);
    }
}
//定义目录过滤器
```

```
class DirFilter implements FileFilter
{
    public boolean accept(File path)
    {
        return path.isDirectory();
    }
}
```

程序运行结果：

```
列出目录 bin\code0803 中的 class 文件
文件:bin\code0803\PrintfDemo.class
文件:bin\code0803\ScannerSystemIn.class
文件:bin\code0803\StdInput.class
列出目录 bin\code0803 中的子目录
true
目录:bin\code0803\_newDir
```

8.4.2 文件字节流

文件字节流类 FileInputStream 和 FileOutputStream 可用于对文件的输入和输出处理。

1. FileInputStream

FileInputStream 用于顺序读取本地文件，从超类继承 read()和 close()等方法。它的两个常用的构造函数如下：

```
FileInputStream(String filepath)
FileInputStream(File fileObj)
```

当指定的文件在文件系统中不存在时，它们都能引发 FileNotFoundException 异常。这里，filepath 是文件的全称路径，fileObj 是描述该文件的 File 对象。假设有一个文件，名称为 Test.java，可以用下面的代码构造文件输入流。

直接使用文件名构造输入流：

```
FileInputStream f1=new FileInputStream("Test.java")
```

或者先构造 File 对象，再用 File 对象构造输入流：

```
File f=new File("Test.java");
FileInputStream f2=new FileInputStream(f);
```

FileInputStream 重写了抽象类 InputStream 的读取数据的方法：

```
public int read()
```

从此输入流中读取一个数据字节。

```
public int read(byte[] b)
```

此输入流中将最多 b.length 个字节的数据读入一个字节数组中。

```
public int read(byte[] b, int off,int len)
```

此输入流中将最多 len 个字节的数据读入一个字节数组中。

这些方法在读取数据时,遇输入流结束则返回－1。

2. FileOutputStream

FileOutputStream 用于向一个文件写数据。它从父类中继承 write(),close()等方法。它常用的构造函数如下:

```
FileOutputStream(String filePath)
FileOutputStream(File fileObj)
FileOutputStream(String filePath, boolean append)
FileOutputStream(File fileObj, boolean append)
```

它们可以引发 IOException 或 SecurityException 异常。这里 filePath 是文件的全称路径,fileObj 是描述该文件的 File 对象。如果 append 为 true,文件以追加方式打开,不覆盖已有文件的内容;如果 append 为 false,则覆盖原文的内容,只有一个参数时,则覆盖文件的内容。

FileOutputStream 的创建不依赖于文件是否存在。如果 filePath 表示的文件不存在,FileOutputStream 在打开之前创建它;如果文件已经存在,则打开它,准备写。如果试图打开一个只读文件,会引发一个 IOException 异常。

FileOutputStream 重写了抽象类 OutputStream 的写数据的方法:

```
public void write(byte[] b)
```

将 b.length 个字节从指定字节数组写入此文件输出流中。

```
public void write(byte[] b, int off, int len)
```

将指定字节数组中从偏移量 off 开始的 len 个字节写入此文件输出流。

```
public void write(int b)
```

将指定字节写入此文件输出流。

b 是 int 类型时,占用 4 个字节,只有最低的一个字节被写到输出流,忽略其余字节。

下面的文件复制程序,使用 FileOutputStream 创建一个输出流,实现源文件到目标文件的内容复制,分别使用了文件流的 3 种重载的 read 和 write 方法,在实际使用过程中,选择一种就可以了。

【例 8.7】 文件复制程序。

```
package code0804;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class FileStreamCopy {
    public static void main(String[] args) throws IOException {
```

```
            int size;
            //构造输入流对象
            FileInputStream f=new FileInputStream("src/code0804/FileStreamCopy.
            java");
            FileOutputStream fout=new FileOutputStream("copy-of-file.txt");
            System.out.println("总长度: "+(size=f.available()));
            int n=size/10;
            System.out.print("使用单字节方法读取后: ");
            //使用 read()和 write
            for (int i=0; i<n; i++) {
                fout.write(f.read());
            }
            System.out.println("剩余长度: "+f.available());
            System.out.print("读取一个字节数组后: ");
            //使用 read(byte[]b)和 write(byte[] b);
            byte b[]=new byte[n];
            f.read(b);
            fout.write(b);
            System.out.println("剩余长度: "+f.available());
            //使用 read(b,offset,len)和 write(b,offset,len)
            System.out.print("读取余下数据: ");
            int count=0;
            while((count=f.read(b, 0, n)) !=-1)
                fout.write(b, 0, count);
            System.out.println("剩余长度: "+f.available());
            //最后注意关闭流
            f.close();
            fout.flush();
            fout.close();
        }
    }
```

程序运行结果：

```
总长度: 1175
使用单字节方法读取后: 剩余长度: 1058
读取一个字节数组后: 剩余长度: 941
读取余下数据: 剩余长度: 0
```

可以用记事本打开 copy-of-file.txt 文件，检查其内容和本程序是相同的。

8.4.3 字节过滤流

过滤流在读写数据的同时可以对数据进行处理，它提供了同步机制，使得某一时刻只有一个线程可以访问一个 I/O 流，以防止多个线程同时对一个 I/O 流进行操作所带来的意想不到的结果。

这些过滤字节流是 FilterInputStream 和 FilterOutputStream。它们的构造函数如下：

```
FilterOutputStream(OutputStream os)
FilterInputStream(InputStream is)
```


为了使用一个过滤流,必须首先把过滤流连接到某个输入输出流上,通过在构造方法的参数中指定所要连接的输入输出流来实现。

过滤流扩展了输入输出流的功能,典型的扩展是缓冲、字符字节转换和数据转换。为了提高数据的传输效率,为一个流配备缓冲区(buffer),所以过滤流有时也称为缓冲流。

当向缓冲流写入数据时,系统将数据发送到缓冲区,而不是直接发送到外部设备,缓冲区自动记录数据,当缓冲区满时,系统将数据全部发送到外部设备。

当从一个缓冲流中读取数据时,系统实际上是从缓冲区中读取数据。当缓冲区空时,系统会自动从相关设备读取数据,并读取尽可能多的数据充满缓冲区。

因为有缓冲区可用,缓冲流支持跳过(skip)、标记(mark)和重新设置流(reset)等方法。

常用的缓冲输入流有 BufferedInputStream、DataInputStream,常用的缓冲输出流有 BufferedOutputStream、DataOutputStream、PrintStream。

缓冲流的操作方法和普通流的操作方法类似,主要使用其 read()方法和 write()方法。下面以 BufferedInputStream 和 BufferedOutputStream 为例进行说明。

1. BufferedInputStream/BufferedOutputStream

缓冲输入/输出是一个非常普通的性能优化。Java 中 BufferedInputStream 类允许把任何字节输入流类"包装"成缓冲流并使它的性能提高。BufferedInputStream 有两个构造函数:

```
BufferedInputStream(InputStream inputStream)
BufferedInputStream(InputStream inputStream, int bufSize)
```

第一种形式生成了一个默认缓冲区长度的缓冲流。第二种形式缓冲区大小是由 bufSize 传入的。使用内存页或磁盘块等的若干倍的缓冲区大小可以给执行性能带来很大的改善,但这是依赖于执行的情况。最理想的缓冲长度一般与主机操作系统、可用内存空间及机器配置有关。合理利用缓冲不需要特别复杂的操作,一般缓冲大小为 8192B。用这样的方法,低级系统可以从磁盘或网络读取数据块并在缓冲区中存储结果。

BufferedOutputStream 用一个 flush()方法来保证数据缓冲区被写入到实际的输出设备。因为 BufferedOutputStream 是通过减小系统写数据的时间来提高性能的,可以调用 flush()方法输出缓冲区中待写的数据。

下面是两个常用的缓冲输出流构造函数:

```
BufferedOutputStream(OutputStream outputStream)
BufferedOutputStream(OutputStream outputStream, int bufSize)
```

第一种形式创建了一个使用 512B 缓冲区的缓冲输出流。第二种形式,缓冲区的大小由 bufSize 参数传入。

下面再用缓冲流来实现文件的复制。

【例 8.8】 使用缓冲流的文件复制程序。

```
package code0804;
```

```
import java.io.BufferedInputStream;
import java.io.BufferedOutputStream;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class BufferedStreamCopy {
    public static void main(String[] args) throws IOException {
        int size;
        //构造输入输出流对象
        FileInputStream f=new FileInputStream("src/code0804/BufferedStreamCopy.
        java");
        FileOutputStream fout=new FileOutputStream("copy-of-file.txt");
        //使用缓冲流
        BufferedInputStream bis=new BufferedInputStream(f);
        BufferedOutputStream bos=new BufferedOutputStream(fout);
        System.out.println("开始复制…");
        int n=f.available()/5;
        byte b[]=new byte[n];
        int count=0;
        while((count=bis.read(b, 0, n)) !=-1)
            bos.write(b, 0, count);
        System.out.println("完成复制");
        bis.close();
        bos.flush();
        bos.close();
        f.close();
        fout.flush();
        fout.close();
    }
}
```

当文件比较大时,使用缓冲流能够提高输入输出的效率。

2. DataInputStream/DataOutputStream

DataInputStream 和 DataOutputStream 它们不仅能使用一般的 read()方法读取数据流,一般的 write()方法写数据流,而且能直接读/写各种 Java 基本数据类型,如 boolean、int、float 和 double 等,这些基本数据类型在文件中的表示方式和它们在内存中的一样,无须多余的转换。

8.4.4 实用案例

【例 8.9】 文件加密解密。

给定一个密钥,读取文件内容,加密后,输出到另外一个文件。

```
package code0804;
import java.io.FileInputStream;
import java.io.FileOutputStream;
```

```
import java.io.IOException;
public class EncryptFile {
    public static void main(String[] args) throws IOException {
        byte pwd=123;
        FileInputStream f=new FileInputStream("src/code0804/
        BufferedStreamCopy.java");
        FileOutputStream fout=new FileOutputStream("encrypted.txt");
        System.out.println("开始加密…");
        int n=f.available()/5;
        byte buf[]=new byte[n];
        int count=0;
        while((count=f.read(buf, 0, n)) !=-1) {
            for (int i=0; i<count; i++) {
                buf[i]=(byte) (buf[i] ^ pwd);//密码与值进行异或运算
            }
            fout.write(buf, 0, count);
        }
        System.out.println("完成加密");
        f.close();
        fout.close();
        f=new FileInputStream("encrypted.txt");
        fout=new FileOutputStream("unencrypted.txt");
        System.out.println("开始解密…");
        n=f.available()/5;
        buf=new byte[n];
        count=0;
        while((count=f.read(buf, 0, n)) !=-1) {
            for (int i=0; i<count; i++) {
                buf[i]=(byte) (buf[i] ^ pwd);//密码与值进行异或运算
            }
            fout.write(buf, 0, count);
        }
        System.out.println("完成解密");
        f.close();
        fout.close();
    }
}
```

程序运行结果:

开始加密…
完成加密
开始解密…
完成解密

分析:这里使用文件输入流读取文件内容,然后每个字节和密码进行异或加密。加密

完成，使用文件输出流写入另外一个文件中。解密方法和加密方法相同。可以用记事本打开加密后的文件进行查看，发现无法阅读。

8.5 字符流使用

尽管 Java 字节流提供了处理任何类型数据输入输出操作的功能，但它们不能直接操作字符。一般文本文件适合用字符流，因此提供直接的字符输入输出支持是必要的。

8.5.1 文件字符流

FileReader 类是一个以字符方式读取文件内容的字符输入流，常用的构造函数如下：

```
FileReader(String filePath)
FileReader(File fileObj)
```

当指定的文件不存在时，引发一个 FileNotFoundException 异常。这里，filePath 是一个文件的完整路径，fileObj 是描述该文件的 File 对象。

FileWriter 类是一个以字符方式写文件内容的 Writer 类的子类，最常用的构造函数如下：

```
FileWriter(String filePath)
FileWriter(String filePath, boolean append)
FileWriter(File fileObj)
```

这里，filePath 是文件的完全路径，fileObj 是描述该文件的 File 对象。如果 append 为 true，输出内容附加到文件尾。只有一个参数的构造方法，默认是覆盖文件内容。

FileWriter 类的创建不依赖于文件存在与否。如果文件不存在，则创建文件，然后打开它作为输出。如果试图打开一个只读文件，将引发一个 IOException 异常。

FileReader 重写了抽象类 Reader 的读取数据的方法。

public int read()：读取单个字符。

public int read(char[] b)：将字符读入数组。

public int read(char[] b, int off,int len)：将字符读入数组中的某一部分。

这些方法在读取数据时，遇输入流结束则返回－1。

FileWriter 重写了抽象类 Writer 的写数据的方法。

public void write(char[] b)：写入字符数组。

public void write(char[] b, int off, int len)：写入字符数组的某一部分。

public void write(int b)：写入单个字符。

b 是 int 类型时，占用 4 个字节，要写入的字符包含在给定整数值的 16 个低位中，16 高位被忽略。

下面用一个程序说明 FileReader 和 FileWriter 类的使用。程序用 FileReader 读取文件的内容，然后写入另外一个目标文件，完成文件复制功能。

【例 8.10】 字符流文件复制程序。

```
package code0805;
```

```
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
public class FileReaderWriterDemo
{
    public static void main(String[] args) throws IOException
    {
        //构造输入流对象
        File file=new File("src/code0805/FileReaderWriterDemo.java");
        FileReader f=new FileReader(file);
        FileWriter fout=new FileWriter("copy-of-file.txt");
        System.out.println("当前字符集是: "+f.getEncoding());
        int n=(int) (file.length()/10);
        char b[]=new char[n];//构造字符数组,缓存文件内容
        int count=0;
        System.out.println("开始复制");
        int times=0;
        while((count=f.read(b, 0, n)) !=-1)
        {
            times++;
            fout.write(b, 0, count);}
        System.out.println("结束复制,使用数组复制次数: "+times);
        f.close();
        fout.close();
    }
}
```

以每次读取一个字符数组的方式复制文件。比较源文件和目标文件,内容是相同的。Java 源程序本身是以字符文本文件存放的,所以适合用字符流方式读取。

8.5.2 字节流向字符流的转化

字节流以单个字节为读写单位,字符流以字符为读写单位。一般字符由多个字节组成。InputStreamReader 和 OutputStreamWriter 用来在字节和字符之间作为中介,可以把以字节形式表示的流转化为特定的平台上的字符表示。可以在构造这些流对象时指定字符编码,也可以用当前平台的默认编码。

InputStreamReader 的构造函数如下:

```
public InputStreamReader(InputStream in)
public InputStreamReader(InputStream in, String charsetName)
public InputStreamReader(InputStream in, Charset cs)
public InputStreamReader(InputStream in, CharsetDecoder dec)
```

第一种形式使用当前平台默认的编码,从字节流 in 构造一个字符流对象;第二种形式使用特定的字符集编码,从字节流 in 构造一个字符流对象;第三种方式使用一个字符集对

象 cs 和字节流 in 构造输入字符流;第四种形式使用字符集解码器 dec 和字节流 in 构造输入字符流。

如果使用了不支持的字符集,那么会产生一个 UnsupportedEncodingException 异常。

OutputStreamWriter 的构造函数如下:

```
public OutputStreamWriter(OutputStream out)
public OutputStreamWriter(OutputStream out, String charsetName)
public OutputStreamWriter(OutputStream out, Charset cs)
public OutputStreamWriter(OutputStream out, CharsetEncoder enc)
```

第一种形式使用当前平台默认的编码,从字节流 out 构造一个字符流对象;第二种形式使用特定的字符集编码,从字节流 out 构造一个字符流对象;第三种方式使用一个字符集对象 cs 和字节流 out 构造输出字符流;第四种形式使用字符集编码器 enc 和字节流 out 构造输出字符流。

下面用一个简单例子说明 InputStreamReader 和 OutputStreamWriter 的使用。

【例 8.11】 字节流向字符流的转换,也实现了文件复制功能。

```
package code0805;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
public class StreamToReaderWriter
{
    public static void main(String[] args) throws IOException
    {
        File file=new File("src/code0805/StreamToReaderWriter.java");
        FileInputStream fin=new FileInputStream(file);
        FileOutputStream fout=new FileOutputStream("copy-of-file.txt");
        //把文件输入字节流向字符流转换
        InputStreamReader isr=new InputStreamReader(fin, "GBK");
        //把文件输出字节流向字符流转换
        OutputStreamWriter osw=new OutputStreamWriter(fout, "GBK");
        System.out.println("当前输入流编码是:"
                +isr.getEncoding());
        System.out.println("当前输出流编码是:"
                +osw.getEncoding());
        int n=(int) (file.length()/30);
        char b[]=new char[n];
        System.out.println("复制开始…");
        int count=0;
        while((count=isr.read(b, 0, n)) !=-1)
            osw.write(b, 0, count);
```

```
            isr.close();
            fin.close();
            osw.flush();
            osw.close();
            fout.flush();
            fout.close();
            System.out.println("复制完成。");
        }
    }
```

在从字节流构造字符流时,指定了字符集 GBK,因为在操作系统中,Java 程序文件是按照 GBK 编码存储的,如果 Java 程序文件用 UTF-8 存储,那么应该在构造函数中指定 UTF-8 编码。只有指定了正确的字符集,才能正确地从字节流构造字符流。如果在输出时指定了不同的字符集,在输出文件中会产生乱码。

8.5.3 Scanner 封装字符流

除了前面介绍的字符流外,还有其他类,例如,CharArrayReader、CharArrayWriter、StringReader、StringWriter、PrintWriter 和 PushbackReader 等。这些字符流的详细使用说明请参考 Java 文档,在此不详述。

一般地,名称以 Reader 结尾的类都是字符输入流,其大部分读取数据的方法都来自 Reader 类;名称以 Writer 结尾的类都是字符输出流,其大部分写数据的方法都来自 Writer 类。名称以 Stream 结尾的类,一般都是字节流。

对文本文件内容的解析和计算结果的格式化输出,是程序设计过程中经常碰到的问题。例如,一个 source. txt(如图 8-1(a)所示)文件中存放着两行整数,请分别计算每一行的和,追加到每一行数字的末尾,并输出到文件 dest. txt 中(如图 8-1(b)所示)。



(a) (b)

图 8-1 source. txt 和 dest. txt 中的文本内容

Java 的输入输出流提供了非常强大的功能,使用工具类 java. util. Scanner 和字符输出流 java. io. PrintWriter 可以非常方便地解决这个问题。

PrintWriter 用于向字符(文本)输出流打印对象的格式化表示形式,此类在 PrintStream 中的所有 print 方法中实现。

Scanner 不仅可以读取数字(见 8. 3. 2 节),还是一个可以使用正则表达式来解析字符串的文本扫描器。Scanner 使用分隔符模式将其输入分解为标记,用不同的 next()方法将得到的标记转换为不同类型的值。在读取下一个标记之前可以使用 hasNext()方法检测一下。默认情况下,分隔符模式与空白匹配。

基本思路是先用 Scanner 类和文件字符输入流读取文件中的每一行,再次用 Scanner

扫描解析每一行，得到需要的数据。

【例 8.12】 解析文本文件的数据并格式化输出。

```
package code0805;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.PrintWriter;
import java.util.Scanner;
public class ScannerPrintWriterDemo {
    public static void main(String[] args) {
        Scanner s= null;
        PrintWriter pw= null;
        try {
            //使用 Scanner 解析文件字符输入流的内容
            s= new Scanner(new FileReader("source.txt"));
            //使用 PrintWriter 进行格式化输出
            pw= new PrintWriter("dest.txt");
            while(s.hasNextLine()) {                //判断是否还有未读行
                String strLine= s.nextLine();       //读取一行
                //调用方法求每一行的和
                int sum= getLineSum(strLine);
                //输出每一行
                pw.println(strLine+ "\t"+ sum);
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } finally {                                 //关闭输入流和输出流
            if (s != null)
                s.close();
            if (pw != null)
                pw.close();
        }
    }
    //计算每一行的和的方法
    private static int getLineSum(String strLine) {
        //使用 Scanner 分割每一行为多个整数
        Scanner s= new Scanner(strLine);            //用 Scanner 解析字符串
        int sum= 0;
        while(s.hasNextInt()) {                     //判断是否还有数据
            sum+ = s.nextInt();                     //获得下一个整数
        }
        return sum;
    }
}
```

提示： Scanner 不仅可以读取解析输入流的内容，还可以扫描解析字符串，转换成需要

的数据。

8.5.4 实用案例

【例 8.13】 文本替换。

查找某一个文件中的字符串,全部替换成另外的字符串。

```
package code0805;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.PrintWriter;
import java.util.Scanner;
public class ReplaceText {
    public static void main(String[] args) throws FileNotFoundException {
        File f;
        String filename;
        System.out.println("输入文件名:");
        Scanner console=new Scanner(System.in);
        filename=console.nextLine();
        f=new File(new File("src/code0805"),filename);
        System.out.println("输入要替换的字符串:");
        String  pattern=console.nextLine();
        System.out.println("输入要替换的结果:");
        String  result=console.nextLine();
        PrintWriter pw=new PrintWriter("replace.txt");
        Scanner reader=new Scanner(f);
        while(reader.hasNextLine())
        {
            String s1=reader.nextLine();
            String s2=s1.replaceAll(pattern, result);
            pw.println(s2);
        }
        console.close();
        pw.close();
    }
}
```

程序运行结果:

```
输入文件名:
ReplaceText.java
输入要替换的字符串:
String
输入要替换的结果:
hello
```

分析: 该程序使用 Scanner 分行读取文件,然后进行替换,使用 PrintWriter 进行输出。

打开输出文件，将所有的 String 替换成 hello。

8.6 串 行 化

8.6.1 串行化的概念

对象的寿命通常随着生成该对象的程序的终止而终止。某些时候，需要将对象的状态保存下来，将来需要的时候可以恢复，或者把对象传输到其他地方。

把对象的这种能记录自己的状态以便将来再生的能力称为对象的持续性(persistence)。对象通过写出描述自己状态的数值来记录自己的过程称为对象的串行化(serialization)。

串行化的主要任务是写出对象实例变量的数值。如果实例变量是另一对象的引用，则引用的对象也要串行化。

Java 提供了对象串行化的机制，在 java.io 包中，定义了一些接口和类作为对象串行化的工具。

1. Serializable 接口

只有实现 Serializable 接口的对象才可以被串行化。Serializable 接口没有定义任何成员。它只用来表示一个类可以被串行化。如果一个类可以串行化，它的所有子类都可以串行化。

2. ObjectOutput 接口

ObjectOutput 接口继承 DataOutput 接口并且支持对象串行化。它的 writeObject()方法，可以输出一个对象。所有这些方法在出错情况下引发 IOException 异常。

```
final void writeObject(Object obj)
```

向流中写入对象 obj。

3. ObjectOutputStream 类

ObjectOutputStream 类继承 OutputStream 类并实现 ObjectOutput 接口。它负责向流中写入对象。该类的构造函数如下：

```
ObjectOutputStream (OutputStream out) throws IOException
```

其中，参数 out 是串行化的对象将要写入的输出流。

4. ObjectInput 接口

ObjectInput 接口继承 DataInput 接口。它支持对象反串行化，其 readObject()方法可以反串行化对象。所有这些方法在出错情况下引发 IOException 异常。

```
Object readObject()
```

从流读取一个对象。

5. ObjectInputStream 类

ObjectInputStream 继承 InputStream 类并实现 ObjectInput 接口。ObjectInputStream 负责从流中读取对象。该类的构造函数如下：

```
ObjectInputStream(InputStream in)
```

其中,参数 in 是串行化对象将被读取的输入流。

6. 串行化注意事项

(1) 串行化只能保存对象的非静态成员变量,不保存变量的修饰符。

(2) 对于 transient 关键字,对于某些类型的变量,其状态是瞬时的,无法或无须保存其状态,对于这些变量,可以用 transient 关键字标明。对于一些需要保密的变量,为了保证安全性,不应该串行化,可以在其前面加上 transient 关键字。

8.6.2 实用案例

【例 8.14】 串行化学生对象。

(1) 定义一个可串行化对象。

被串行化的类必须实现 Serializable 接口。

(2) 构造对象输入输出流。

要串行化一个对象,必须与对象的输入输出联系起来,通过 writeObject()串行化对象,通过 readObject()方法反串行化对象。

程序示例如下:

```
package code0806;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
class Student implements Serializable {
    private static final long serialVersionUID=1L;
    int id;
    String name;
    int age;
    String department;
    public Student(int id, String name, int age, String department) {
        this.id=id;
        this.name=name;
        this.age=age;
        this.department=department;
    }
}
public class SerializableDemo {
    public static void main(String args[]) throws Exception {
        Student stu=new Student(20151064, "zhang shan", 22, "CQUCS");
        FileOutputStream fout=new FileOutputStream("data1.ser");
        ObjectOutputStream oout=new ObjectOutputStream(fout);
        //输出对象
        oout.writeObject(stu);
```

```
            oout.close();
            stu=null;
            FileInputStream fin=new FileInputStream("data1.ser");
            ObjectInputStream oin=new ObjectInputStream(fin);
            //读入对象
            stu=(Student)oin.readObject();
            oin.close();
            System.out.println("学生信息：");
            System.out.println("ID:          "+stu.id);
            System.out.println("name:        "+stu.name);
            System.out.println("age:         "+stu.age);
            System.out.println("department: "+stu.department);
        }
    }
```

程序运行结果：

```
学生信息：
ID:          20151064
name:        zhang shan
age:         22
department: CQUCS
```

分析：先构造一个文件流，然后再构造对应的对象流。对象 stu 先被串行化到文件 data1.ser 中，然后读入内存再生。从输出结果可以看出，通过串行化机制正确地保存和恢复了对象的状态。

8.7 输入输出处理实训任务

【任务描述】

编写一个文件分割与合并程序，把一个大的原始文件分割成多个指定大小的小文件，并且能够把分割后的小文件合并成原始文件。

【任务分析】

要求按照指定大小对文件进行分割，那么首先需要获得被分割文件的大小。其次，根据分割后每个文件的大小，可以计算出分割后文件的数量。对分割后的文件进行编号，这样便于对文件进行合并。

可以定义两个函数：一个进行文件分割，另一个进行文件合并。需要使用文件输入输出流来读写文件内容。

设创建的程序文件名为 FileCutMerge.java。可以定义两个命令行参数，如-c4000 filename 表示分割文件，每个文件大小 4000B；-m prefix 表示把当前目录下文件名前传为 prefix 的文件合并为一个文件。

程序的基本工作流程如下：

(1) 分析命令行参数，决定是分割文件还是合并文件。

（2）调用具体的函数进行分割和合并。

【任务解决】

完整程序如下：

```
import java.io.File;
import java.io.FileFilter;
import java.io.FileInputStream;
import java.io.FileOutputStream;
public class FileCutMerge {
    //分析命令行参数,决定是分割文件还是合并文件
    public static void main(String[] args) {
        //创建一个文本类的对象
        FileCutMerge tool=new FileCutMerge();
        //判断命令行参数的数量
        if ((args==null) || (args.length != 2)) {
            tool.help();
        } else if (args[0].startsWith("-c")) {
            //被分割文件的文件名,从当前目录开始
            File f1=new File(args[1]);
            if(!f1.exists()) {
                System.out.println("指定的文件不存在");
            }
            //从命令行参数获得分割后文件的大小
            int fileSize=Integer.parseInt(args[0].substring(2));
            try {
                //分割文件
                tool.cut(f1, fileSize);
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else if (args[0].equals("-m")) {
            //被合并的小文件名的前缀,就是-前面的内容
            String prefixname=args[1];
            File f=new File(".");
            //列出当前目录下需要合并的文件,使用了一个文件过滤器
            File[] names=f.listFiles(new MyFilesFilter(prefixname));
            try {
                //合并文件
                tool.merge(names);
            } catch (Exception e) {
                e.printStackTrace();
            }
        } else {
        }
    }
```

```
public void help() {
    System.out.println("错误的命令行格式,正确的是：");
    System.out.println("FileCutMerge -c filename");
    System.out.println("或者");
    System.out.println("FileCutMerge -m filenameprefix");
}
public void cut(File file, int size) throws Exception {
    System.out.println("开始分割文件…");
    //获得被分割文件的父目录,把分割后的文件放到该目录下
    File parent=file.getParentFile();
    long fileLength=file.length();//获得文件大小
    //获得分割后小文件的数目
    int filenum=(int) (fileLength/size);
    if (fileLength%size !=0) {
        filenum+=1;
    }
    String[] smallfilenames=new String[filenum];
    //创建文件输入流读取被分割的文件
    FileInputStream fin=new FileInputStream(file);
    //构造一个字节数组,每次读取一个字节数组的数据
    byte[] buf=new byte[size];
    for (int i=0; i<filenum; i++) {
        //构造分割后的文件名
        File outfile=new File(parent, file.getName()+"-"+i);
        //构造文件输出流
        FileOutputStream fout=new FileOutputStream(outfile);
        //读取数据
        int count=fin.read(buf);
        //输出数据
        fout.write(buf, 0, count);
        fout.close();
        smallfilenames[i]=outfile.getName();
    }
    fin.close();
    //输出分割后的文件名
    System.out.println("分割后的文件如下：");
    for (int i=0; i<smallfilenames.length; i++) {
        System.out.println(smallfilenames[i]);
    }
    System.out.println("文件分割完成。");
}
public void merge(File[] files) throws Exception {
    System.out.println("开始合并文件…");
    //获得目标文件名,来源于被合并的文件
    String smallfilename=files[0].getName();
```

```
            int pos=smallfilename.indexOf("-");
            String tagetfilename="new-"+smallfilename.substring(0, pos);
            System.out.println("合并后的文件为："+tagetfilename);
            File outFile=new File(files[0].getParentFile(), tagetfilename);
            FileOutputStream fout=new FileOutputStream(outFile);
            //合并文件内容,输出到目标文件
            for (int i=0; i<files.length; i++) {
                FileInputStream fin=new FileInputStream(files[i]);
                int b;
                while((b=fin.read()) !=-1) {
                    fout.write(b);
                }
                fin.close();
            }
            fout.close();

            System.out.println("合并文件完成。");
        }
    }

    //文件过滤器,列出目录下符合条件的文件,此类是根据文件名的前缀进行过滤
    class MyFilesFilter implements FileFilter
    {
        String prefix="";
        public MyFilesFilter(String prefix) {
            this.prefix=prefix;
        }
        @Override
        public boolean accept(File f) {
            if (f.getName().length()>prefix.length()
                    && f.getName().startsWith(prefix)) {
                return true;
            }
            return false;
        }

    }
```

假设把文件 FileCutMerge.java 移动到系统当前目录下。分割文件的命令行如下：

```
java.exe FileCutMerge -c500 FileCutMerge.java
```

表示把 FileCutMerge.java 分割为大小为 500 的小文件。程序运行结果如下：

```
开始分割文件…
分割后的文件如下：
FileCutMerge.java-0
```

```
FileCutMerge.java-1
FileCutMerge.java-2
FileCutMerge.java-3
FileCutMerge.java-4
FileCutMerge.java-5
FileCutMerge.java-6
FileCutMerge.java-7
文件分割完成。
```

合并文件的命令行如下：

```
java.exe FileCutMerge -m FileCutMerge.java
```

表示把当前目录下文件名前缀为 FileCutMerge.java 的文件合并为一个大文件。程序运行结果如下：

```
开始合并文件…
合并后的文件为：new-FileCutMerge.java
合并文件完成。
```

习题与思考

1. 简述用哪几种方法可以对文件进行读写。
2. 使用 File 类列出某一个目录下创建日期晚于 2015-8-12 的文件。
3. 使用 File 类创建一个多层目录 D:\java\my Program。
4. 从字节流到字符流的转化过程中，有哪些注意事项？
5. 读取一个 Java 源程序，找出其中使用到的关键字，并统计其个数。

第 9 章 Java 多线程

本章学习目标

线程也称为轻型进程，是处理器调度的基本单位，灵活使用多线程进行程序设计，可以有效地简化设计任务，提高程序执行效率。例如，当使用浏览器下载某个文件的时候，通常只有一个线程，因此效率较低。当改用迅雷等支持多线程的专用软件后，文件的下载任务被分解成多个线程并发执行，下载速度和效率明显得到了提高。

本章将主要介绍线程的基本概念、创建线程的方法、线程的生命周期及调度、线程的同步、线程池的使用等。通过本章的学习，应该重点掌握以下主要内容：

(1) 线程的创建。

(2) 线程的调度。

(3) 线程的同步。

(4) 线程池的使用。

9.1 为什么使用多线程

前面写的 Java 程序在一个时间段内只能完成一个任务，程序代码主要是顺序执行。有的时候，需要一个 Java 程序在一段时间内并发完成多个任务，这就需要使用多线程。例如，对从 1 到 1000 的数求和、求平方和。普通的 Java 程序可以串行执行这两个任务，使用多线程后，可以让一个线程执行求和任务，另一个线程执行求平方和的任务，两个线程并发执行。

【例 9.1】 使用两个线程分别执行数的求和与平方和运算。

分析：创建两个线程，一个是 SumThread，求和；另一个是 SquareSumThread，求平方和。Java 中创建线程最简单的方法就是继承一个特殊的父类 Thread。

问题解决如下：

```
package code0901;
public class SimpleThreadTest {
    public static void main(String[] args) {
        //创建线程对象：
        Thread sum=new SumThread();
```

```
            Thread squareSum=new SquareSumThread();
            //启动线程
            sum.start();
            squareSum.start();
        }
    }
    //继承 Thread创建线程
    class SumThread extends Thread {
        public void run() {
            int sum=0;
            for (int i=1; i<=1000; i++) {
                sum+=i * i;
            }
            System.out.println("平方和:"+sum);
        }
    }
    //继承 Thread创建线程
    class SquareSumThread extends Thread {
        public void run() {
            int sum=0;
            for (int i=1; i<=1000; i++) {
                sum+=i;
            }
            System.out.println("和:"+sum);
        }
    }
```

程序运行结果：

```
平方和：333833500
和：500500
```

该问题的求解涉及以下几个关键点：

(1) 通过继承父类 Thread，创建自定义线程，本例创建了两个线程 SumThread 和 SquareSumThread。关于线程创建的方法，参见 9.3 节。

(2) 线程的核心代码在 run()方法中，也被称为线程体，具体参见 9.3 节。

(3) 创建线程对象后，run()方法不会自动执行，需要调用 start()方法启动线程，具体参见 9.3 节。

(4) 多线程的执行是并发的，独立线程之间没有确定的顺序关系，输出结果的先后顺序因此不确定。具体原因参见 9.4.2 节。

9.2 线程的概念

一般每个程序都有一个入口、一个出口以及一个顺序执行的序列，执行中的程序称为进程，在进程执行过程中的任何指定时刻，都只有一个单独的执行点。在多线程情况下，在单

个进程内部可以在同一时刻进行多种运算,有多个执行点。

一个单独的线程和进程相似,也有一个入口、一个出口以及一个顺序执行的序列,从概念上说,一个线程是一个进程内部的一个顺序控制流,必须在进程中运行。在一个进程中可以实现多个线程,这些线程同时运行,完成不同的功能。从逻辑的观点来看,多线程意味着一个程序的多行语句同时执行,操作系统不会把每个线程当作独立的进程来对待。

线程与进程的比较如下:

(1) 两者的粒度不同。进程是由操作系统来管理的,而线程则是在一个程序(进程)内。

(2) 不同进程的代码、内部数据和状态都是完全独立的,而一个进程内的多线程是共享进程的内存空间和系统资源,有可能互相影响。

(3) 线程本身的数据通常只有寄存器数据以及一个程序执行时使用的堆栈,所以线程的切换比进程切换的负担要小。

使用多线程具有如下优点:

(1) 多线程编程简单,效率高(线程间能直接共享数据和资源,而多进程不能)。

(2) 适合于开发服务程序(如 Web 服务、聊天服务等)。

(3) 适合于开发有多种交互接口的程序(如聊天程序的客户端、网络下载工具)。

(4) 适合于有人机交互又有计算量的程序(如字处理程序 Word、Excel)。

(5) 程序的吞吐量会得到改善(同时监听多种设备,如网络端口、串口、并口以及其他外设)。

(6) 有多个处理器的系统,可以并发运行不同的线程。

9.3 线程的创建

Java 提供了类 java.lang.Thread 来支持多线程编程,这个类提供了大量的方法控制线程。创建一个线程就是创建 Thread 类或者它的子类的对象。Thread 类有很多重载的构造方法:

```
Thread()
Thread(Runnable target)
Thread(Runnable target, String name)
Thread(String name)
Thread(ThreadGroup group, Runnable target)
Thread(ThreadGroup group, Runnable target, String name)
Thread(ThreadGroup group, String name)
```

参数 target 是线程执行的目标对象,即线程执行的代码,一般是实现了 java.lang.Runnable 接口的对象;group 是线程所在的组;name 是线程的名字。

Java 中创建线程主要有两种方法:一种是继承 Thread;另一种是实现接口 Runnable。可以根据具体的应用环境进行选择。

9.3.1 采用继承创建线程

采用继承创建线程的方法比较简单,主要是通过继承 java.lang.Thread 类,并重写

Thread 类的 run()方法来完成线程的创建。Thread 类封装了线程的行为。要创建一个线程,可以创建一个 Thread 类的子类。Thread 类中有两个最重要的方法 run()和 start()。

创建线程时,run()方法必须被重写,以将线程所要执行的代码加入到这个方法,也就是加入线程体中。虽然 run()方法是线程体,但不能直接调用 run()方法,而是通过调用 start()方法来启动线程。在调用 start()的时候,start()方法会首先进行与多线程相关的初始化(这也是为什么不能直接调用 run()方法的原因),然后再调用 run()方法。

下面是一个示例程序。

【例 9.2】 使用继承方法创建线程的例子。

```
package0903;
//继承 Tread类
public class MyThread extends Thread
{ //count 变量用于统计打印的次数并共享变量
    private static int count=0;
    public MyThread(String name)
    {
        super(name);
    }
    public static void main(String[] args)
    {//main 方法开始
        MyThread p=new MyThread("t1");        //创建一个线程实例
        p.start();                            //执行线程
        //主线程 main 方法执行一个循环
        for (int i=0; i<5; i++) {
            count++;
            //主线程中打印 count+"main"变量的值,并换行
            System.out.println(count+": main");
        }
    }
    public void run()
    {//线程类必须有的 run()方法
        for (int i=0; i<5; i++) {
            count++;
            System.out.println(count+":"+this.getName());
        }
    }
}
```

程序运行结果:

```
1: main
3: main
4: main
5: main
6: main
2:t1
```

```
7:t1
8:t1
9:t1
10:t1
```

分析：上面这段程序用 Java 虚拟机启动程序后，main 方法生成新线程 t1，并通过 for 循环输出变量 count 的值和线程的名称。main 线程和 t1 线程都在操作变量 count。

9.3.2 通过实现接口创建线程

Java 中只允许单继承，如果类已经继承了其他的类，那么就无法再继承 Thread 类了。为此，Java 中提供了另外一种方法来实现多线程。这种方法通过实现 java.lang.Runnable 接口创建多线程。该接口只定义了一个方法 run()，所以必须在新类中实现它。但是，Runnable 接口并没有任何对线程的支持，还必须创建 Thread 类的对象，然后把 Runnable 对象传递给 Thread 对象，这可以通过 Thread 类的构造方法 public Thread(Runnable target)来实现。

下面是使用这一方法的示例程序。

【例 9.3】 使用接口创建线程的例子。

```
package code 0903;
//实现 Runnable 接口
public class MyThread2 implements Runnable
{
    int count=1, number;
    public MyThread2(int i)
    {
        number=i;
        System.out.println("创建线程 "+number);
    }
    public void run()
    {
        while(true) {
            System.out.println("线程 "+number+":计数 "+count);
            if (++count==6)
                return;
            {
            }
        }
    }
    public static void main(String args[])
    {
        for (int i=0; i<5; i++)
            //通过 Thread类创建线程对象,并启动
            new Thread(new MyThread2(i+1)).start();
    }
}
```

程序运行结果：

```
创建线程 1
创建线程 2
创建线程 3
线程 1:计数 1
线程 1:计数 2
线程 1:计数 3
线程 1:计数 4
线程 1:计数 5
创建线程 4
创建线程 5
线程 2:计数 1
线程 2:计数 2
线程 2:计数 3
...
线程 3:计数 3
线程 3:计数 4
线程 3:计数 5
```

分析：由于多线程之间执行顺序的不确定性，每次执行程序的输出结果可能不同。这种创建线程的方式是比较灵活的方式，也是常用的创建线程的方式。其特点是，专门创建一个类实现 Runnable 接口的 run()方法，提供线程的执行代码作为线程体；然后，再创建一个线程对象去执行这个线程体，实现了线程代码和线程对象的分离。

9.3.3 实用案例

【例 9.4】 使用线程池创建线程。

前面章节讲解的两种创建线程的方法都需要新建 Thread 对象，这些线程完成任务后就死亡。有时需要使用已经存在的线程对象，反复执行特定的任务。这种情况下，可以使用线程池。其基本思想是：事先创建一些待命的线程，如果需要执行任务，则从线程池中取一个线程来执行指定的任务，任务完成后，把线程放回线程池。可以重复使用线程，避免了重复创建线程对象，节省了资源。

本例使用固定线程池创建线程，然后让它们执行特定的任务。

程序代码如下：

```java
package code0903;
import java.util.Random;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
public class ThreadPoolTarget implements Runnable{
    private String targetName;
    Random r= new Random();
    public ThreadPoolTarget(String targetName) {
        this.targetName= targetName;
```

```
        }
        @Override
        public void run() {
            int sum=0;
            for(int i=0;i<100;i++)
            {
                sum+=r.nextInt(100);
            }
            System.out.println(Thread.currentThread().getName()+"执行:"+
    this.targetName+"结果:"+sum);
        }
        public static void main(String[] args) {
            //创建线程池包含 2个线程
            ExecutorService pool=Executors.newFixedThreadPool(2);
            //创建 10个线程任务,并提交到线程池
            for(int i=1;i<=10;i++)
            {
                //创建线程执行任务
                ThreadPoolTarget t=new ThreadPoolTarget("Target"+i);
                //把任务提交到线程池
                pool.execute(t);
            }
        }
    }
```

程序运行结果:

```
pool-1-thread-1执行:Target1 结果:4925
pool-1-thread-2执行:Target2 结果:5023
pool-1-thread-2执行:Target4 结果:4842
pool-1-thread-1执行:Target3 结果:5065
pool-1-thread-2执行:Target5 结果:4656
pool-1-thread-1执行:Target6 结果:5058
pool-1-thread-2执行:Target7 结果:4940
pool-1-thread-1执行:Target8 结果:5184
pool-1-thread-2执行:Target9 结果:4978
pool-1-thread-1执行:Target10 结果:5004
```

分析:从输出结果可以看出,线程池中的两个线程对象实现了重用,完成了 10 个任务。线程池的其他更深入用法,请参考相关技术文档。

9.4 线程的生命周期及调度

9.4.1 线程生命周期

线程是动态的,具有一定的生命周期,分别经历从创建、执行、阻塞直到消亡的过程。在

每个线程类中都定义了用于完成实际功能的 run 方法，这个 run 方法称为线程体。按照线程体在计算机系统内存中的状态不同，可以将线程分为创建(new)、就绪(runnable)、阻塞(blocked)和死亡(dead)4 个状态，各个状态之间的状态转换过程如图 9-1 所示。

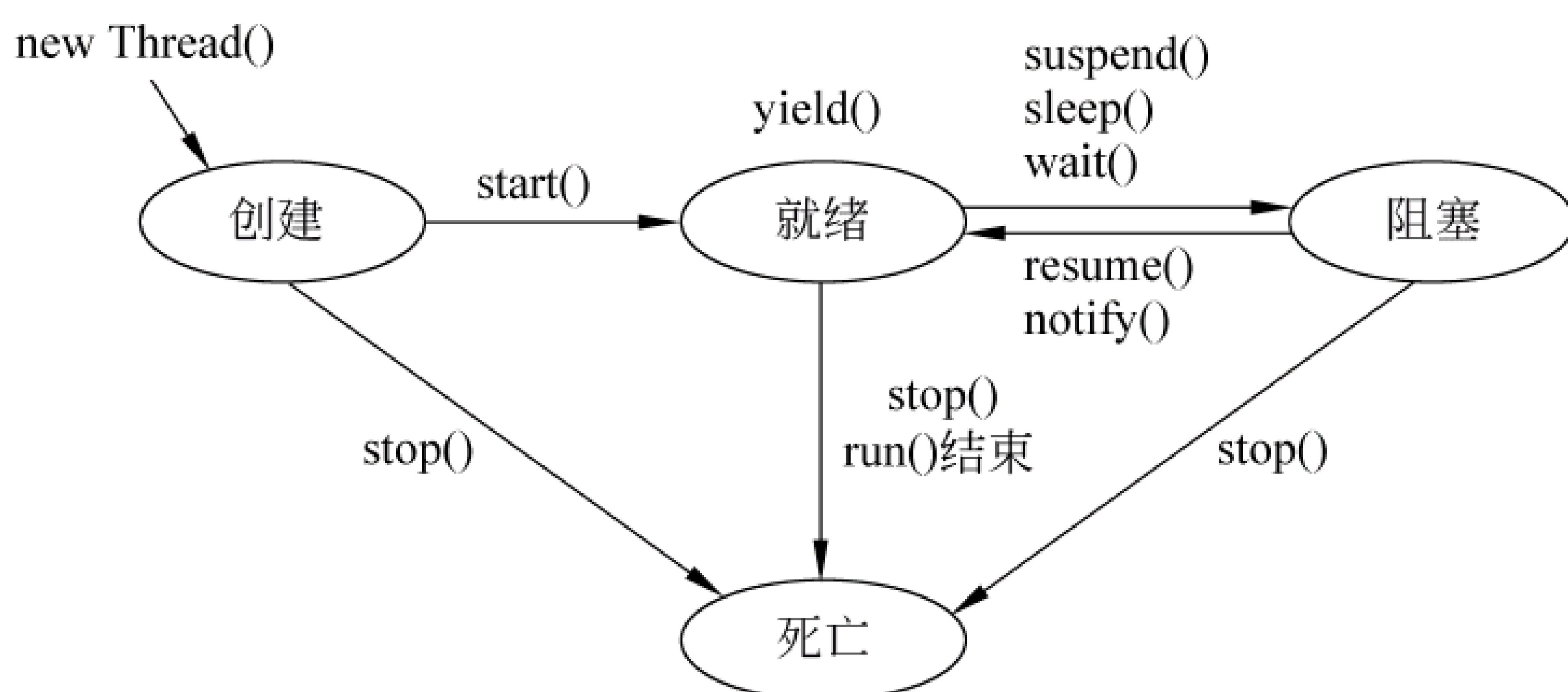


图 9-1　线程状态转换图

1. 创建状态

当利用 new 关键字创建线程对象实例后，它仅仅作为一个对象实例存在，JVM 没有为其分配 CPU 时间片等线程运行资源。

2. 就绪状态

处于创建状态的线程调用 start()方法启动线程，将线程的状态转换为就绪状态(也称为可运行态)。这时，线程已经得到除 CPU 时间之外的其他系统资源，只等 JVM 的线程调度器对该线程进行调度，从而使该线程拥有能够获得 CPU 时间片的机会，一旦该线程获得 CPU 就进入运行状态(本书为后文表达方便，将运行状态并入就绪状态)。运行的线程可以调用 yield()自动放弃 CPU，从而回到就绪状态，以便其他线程运行。

3. 阻塞状态

阻塞指的是暂停一个线程的执行以等待某个条件发生(如某资源就绪)，若线程处于阻塞状态，调度机制不给它分配任何 CPU 时间，直接跳过它。

4. 死亡状态

当线程体运行结束或者调用线程对象的 stop 方法后，线程将终止运行，由 JVM 收回线程占用的资源。

在状态转换的各个过程中，最关键也是最复杂的就是就绪状态和阻塞状态转换的过程。Java 提供了大量方法来支持阻塞，下面逐一分析。

1) sleep(long n)方法

sleep()允许指定以毫秒为单位的一段时间作为参数，它使得线程在指定的时间周期内进入阻塞状态，不能得到 CPU 时间，指定的时间一过，线程重新自动进入可执行状态。典型地，sleep()被用在等待某个资源就绪的情形：测试发现条件不满足后，让线程阻塞一段时间后重新测试，直到条件满足为止。

2) suspend()和 resume()方法

两个方法配套使用，suspend()使得线程进入阻塞状态，并且不会自动恢复，必须其对应的 resume()方法被调用，才能使得线程重新进入可执行状态。suspend()和 resume()被用在等待另一个线程产生结果的情形：测试发现结果还没有产生后，让线程阻塞，另一个线程

产生了结果后,调用 resume()使其恢复。

在线程运行过程中调用 sleep()方法后,该线程在不释放占用资源的情况下,停止运行指定的睡眠时间。时间到达后,线程重新由 JVM 线程调度器进行调度和管理。而调用 suspend()方法后,线程将释放占用的所有资源,由 JVM 调度转入临时存储空间,直至应用程序调用 resume 方法恢复线程运行。

3) wait()和 notify()方法

两个方法配套使用,wait()使得线程进入阻塞状态,它有两种形式:一种允许指定以毫秒为单位的一段时间作为参数,另一种没有参数,前者当对应的 notify()被调用或者超出指定时间时线程重新进入就绪,后者则必须对应的 notify()被调用。这两个方法在线程的同步与互斥中使用。

9.4.2 线程调度和优先级

虽然说线程是并发运行的,然而事实常常并非如此。当系统中只有一个 CPU 时,以某种顺序在单 CPU 情况下执行多线程称为调度(scheduling)。Java 采用的是基于优先级的调度方法。这种算法是根据处于可运行态线程的相对优先级来实行调度。当线程产生时,它继承原线程的优先级。在需要时可对优先级进行修改。在任何时刻,如果有多个线程等待运行,系统选择优先级最高的可运行线程运行。只有当它停止、自动放弃或由于某种原因成为非运行态,低优先级的线程才能运行。如果两个线程具有相同的优先级,它们将被交替地运行。Java 实时系统的线程调度算法还是强制性的,在任何时刻,如果一个比其他线程优先级都高的线程的状态变为可运行态,实时系统将选择该线程来运行。

Java 将线程的优先级分为 10 个等级,分别用 1~10 之间的数字表示。数字越大表明线程的级别越高。相应地,在 Thread 类中定义了表示线程最低、最高和普通优先级的成员变量 MIN_PRIORITY、MAX_PRIORITY 和 NORMAL_PRIORITY,代表的优先级等级分别为 1、10 和 5。当一个线程对象被创建时,其默认的线程优先级是 5。

在应用程序中设置线程优先级的方法很简单,在创建线程对象之后可以调用线程对象的 setPriority()方法改变该线程的运行优先级,同样可以调用 getPriority()方法获取当前线程的优先级。

Java 中有一种比较特殊的线程称为守护(Daemon)线程。这个线程具有最低的优先级,用于为系统中的其他对象和线程提供服务。将一个用户线程设置为守护线程的方式是在线程对象启动之前调用线程对象的 setDaemon()方法。典型的守护线程例子是 JVM 中的系统资源自动回收线程,它始终在低级别的状态中运行,用于实时监控和管理系统中的可回收资源。

【例 9.5】 线程优先级。

```
package code0904;
public class TestThreadPriority extends Thread{
    public TestThreadPriority(String name)      {
        super(name);
    }
    public static void main(String[] args)
```

```
    {
        TestThreadPriority t1=new TestThreadPriority("Thread1");
        t1.setPriority(Thread.MIN_PRIORITY);
        t1.start();
        TestThreadPriority t2=new TestThreadPriority("Thread2");
        t2.setPriority(Thread.NORM_PRIORITY);
        t2.start();
        TestThreadPriority t3=new TestThreadPriority("Thread3");
        t3.setPriority(Thread.MAX_PRIORITY);
        t3.start();
    }
    public void run()
    {
        for (int i=0; i<5; i++)
            System.out.println(this.getName()+" is running!");
    }
}
```

程序运行结果：

```
Thread1 is running!
Thread2 is running!
Thread3 is running!
Thread3 is running!
Thread2 is running!
Thread1 is running!
Thread2 is running!
Thread2 is running!
Thread3 is running!
Thread3 is running!
Thread3 is running!
Thread2 is running!
Thread1 is running!
Thread1 is running!
Thread1 is running!
```

分析：由程序运行结果可以看出，高优先级的线程有优先执行的机会，低优先级的线程也有执行的机会，不一定是高优先级执行完成后它才执行。

9.4.3 线程的终止

在前面的程序中，线程体 run()方法执行完成后，线程就终止了，也可以用修改某些变量以指示目标线程应该停止运行的代码来取代。要使线程在完成任务之前可取消，必须采取一定的措施，但应该是一个清晰而安全的机制使线程终止。

Thread 类提供了 stop()方法，任何时候，调用线程的 stop()方法可以终止线程，但是这是一个不安全的方法，会破坏线程的状态，已经不推荐使用。

另外一种方法是通过中断线程来请求取消线程的执行,并且让线程来监视并响应中断。中断请求通常是用户希望能够终止线程的执行,但并不会强制终止线程,但是它会中断线程的睡眠状态,例如调用 sleep()和 wait()方法后。

Thread 类提供的中断线程的方法有:interrupt(),向线程发送中断;isInterrupted(),测试线程是否已经被中断;Interrupted(),测试当前线程是否已经被中断,随后清楚线程"中断"状态的静态方法。

线程的中断状态只能由线程自己清除,当线程侦测到自己被中断时,经常需要在响应中断之前做某些清除工作,这些清除工作可能涉及那些在线程仍然保持中断状态时会受到影响的操作。

如果被中断的线程正在执行 sleep()或者 wait()方法,就会抛出 InterruptedException 异常。这种抛出异常的中断会清除线程的中断状态。

大体上任何执行阻塞操作的方法,都应该通过 Interrupt 来取消阻塞操作。

下面的程序,主线程在等待计算线程 2000ms 后,中断计算线程,计算线程由于正在执行 sleep(),就会抛出 InterruptedException 异常,终止休眠状态,然后进入异常处理,在 catch 中可以做一些清理工作(如果需要),然后线程执行结束。

【例 9.6】 中断线程的执行。

```
package code0904;
public class InterruptTest extends Thread {
    static int result=0;
    public InterruptTest(String name) {
        super(name);
    }
    public static void main(String[] args) {
        System.out.println("主线程执行");
        Thread t=new InterruptTest("计算线程");
        t.start();
        System.out.println("result: "+result);
        try {
            long start=System.nanoTime();     //获得开始时间
            t.join(2000);                     //等待线程 t 执行 2000ms
            long end=System.nanoTime();       //获得结束时间
            t.interrupt();                    //中断线程 t 的执行
            System.out.println((end-start)/1000000+"毫秒后:"+result);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    @Override
    public void run() {
        System.out.println(this.getName()+"开始计算…");
        try {
            Thread.sleep(4000);
```

```
            } catch (InterruptedException e) {
                System.out.println(this.getName()+"被中断,结束");
                return;
            }
            result= (int) (Math.random() * 10000);
            System.out.println(this.getName()+"结束计算");
        }
    }
```

程序运行结果：

```
主线程执行
result: 0
计算线程开始计算…
1999毫秒后:0
计算线程被中断,结束
```

分析：从运行结果可以看出，计算线程被中断后，run()方法中的最后两行语句没有执行。没有产生计算结果。程序中的join()方法，也是一个控制线程执行的方法，表示一个线程等待另一个线程执行一段时间之后执行，或者等待另一个线程执行结束之后执行。

main作为主线程，在main中调用t.join()，表示main线程等待t线程结束之后再执行。t.join(2000)，表示main线程等待t线程2000ms之后再执行。

如果一个线程长时间没有调用能够抛出InterruptedException异常的方法，那么线程就必须定期地调用Thread.interrupted方法，若接收到中断则返回true，然后就可以退出线程。

9.4.4 实用案例

【例9.7】 向线程发送中断信号。

一般强行终止线程，有可能破坏线程所操作数据的一致性；应该向线程发送中断信号，线程收到中断信号后，自行执行结束操作。

```
package code0904;
public class InterruptTest2 extends Thread {
    static int result=0;
    public InterruptTest2(String name) {
        super(name);
    }
    public static void main(String[] args) {
        System.out.println("主线程执行");
        Thread t=new InterruptTest2("计算线程");
        t.start();
        System.out.println("result: "+result);
        try {
            long start=System.nanoTime();
            t.join(10);
```

```
                long end= System.nanoTime();
                t.interrupt();
                System.out.println((end- start)/1000000+ "毫秒后:"+ result);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        @Override
        public void run() {
            System.out.println(this.getName()+ "开始计算…");
            for (int i= 0; i< 1000000; i+ + ) {
                result+ + ;
                if (Thread.interrupted()) {
                    System.out.println(this.getName()+ "被中断,即将结束");
                    return;
                }
            }
            System.out.println(this.getName()+ "结束计算");
        }
    }
```

程序运行结果:

```
主线程执行
result: 0
计算线程开始计算…
9毫秒后:786688
计算线程被中断,即将结束
```

分析:上面的程序,计算线程原计划执行 1000000 次循环,主线程等待 9ms 后,中断计算线程,计算线程接收到中断后就结束执行,得到当前计算结果。

9.5 多线程互斥与同步

9.5.1 线程的互斥

1. 问题的提出

已知一个银行账号,当从多个渠道同时取钱的时候,有可能造成账号相关数据被破坏,请看下面的例子。

【例 9.8】 错误的银行账号对象。

```
package code0905;
public class WrongAccount {
    double balance;                //账号余额
    public WrongAccount(double balance) {
        super();
```

```
            this.balance=balance;
        }
}
class AccountThread extends Thread
{
    WrongAccount account;
    int delay=100;
    public AccountThread(WrongAccount account) {
        super();
        this.account=account;
    }
    @Override
    public void run() {
        //下面执行取款操作
        if(account.balance>=1000)
        {
            try {
                sleep(delay);
                account.balance-=1000;
                System.out.println("取款 1000 成功。");
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }else
        {
            System.out.println("取款失败。");
        }
    }
    public static void main(String[] args) throws InterruptedException
    {
        //创建账号对象
        WrongAccount account=new WrongAccount(1005);
        //创建 3个取款线程
        AccountThread t1=new AccountThread(account);
        AccountThread t2=new AccountThread(account);
        AccountThread t3=new AccountThread(account);
        t1.start();
        t2.start();
        t3.start();
        //主线程等待 3个线程结束后,输出最后余额
        t1.join();
        t2.join();
        t3.join();
        System.out.println("最终账号余额是:"+account.balance);
    }
}
```

程序第一次执行结果如下：

```
取款 1000 成功。
取款 1000 成功。
取款 1000 成功。
最终账号余额是：5.0
```

程序再次执行结果如下：

```
取款 1000 成功。
取款 1000 成功。
取款 1000 成功。
最终账号余额是：-995.0
```

分析：从输出结果可以看出，当多线程同时访问一个账号对象时，输出结果是不确定的。3 个线程都取款成功。正确的情况应该是只有一个线程能够取款成功，说明本程序在多线程并行的时候是有问题的。原因在于，当一个线程对数据进行修改后，另外的线程并不知道。多线程并发访问账号对象时，应该进行保护，同一时刻，只能有一个线程访问对象，以保证对象状态的正确性。

2. 互斥对象

通常把多线程并发访问的资源称为临界资源。对临界资源的访问必须是互斥的，Java 可以为每个对象设置一个“互斥锁”，保证同一时刻只有一个线程拥有互斥锁。其他线程必须等待拥有锁的线程释放后才可以获取。

Java 提供了关键字 synchronized 来实现互斥锁。当定义类、方法或者代码片段中，使用该关键字，就表示和该关键字相关联的对象有互斥锁。下面修改例 9.8 的程序，实现多线程互斥。

【例 9.9】 正确银行账号对象。

```
package code0905;
public class RightAccount {
    double balance;//账号余额
    public RightAccount(double balance) {
        super();
        this.balance=balance;
    }
    //同步方法,执行该方法时,必须获得所在对象的互斥锁
    public synchronized void withdraw(double money) {
        //下面执行取款操作
        if (balance>=1000) {
            try {
                Thread.sleep(100);
                balance -=1000;
                System.out.println("取款 1000 成功。");
            } catch (Exception e) {
                e.printStackTrace();
```

```
                }
            } else {
                System.out.println("取款失败。");
            }
        }
    }
    class AccountThread2 extends Thread {
        RightAccount account;
        int delay=100;
        public AccountThread2(RightAccount account) {
            super();
            this.account=account;
        }
        @Override
        public void run() {
            account.withdraw(1000);
        }
        public static void main(String[] args) throws InterruptedException {
            //创建账号对象
            RightAccount account=new RightAccount(1005);
            //创建 3个取款线程
            AccountThread2 t1=new AccountThread2(account);
            AccountThread2 t2=new AccountThread2(account);
            AccountThread2 t3=new AccountThread2(account);
            t1.start();
            t2.start();
            t3.start();
            //主线程等待 3个线程结束后,输出最后余额
            t1.join();
            t2.join();
            t3.join();
            System.out.println("最终账号余额是:"+account.balance);
        }
    }
```

程序的运行结果:

```
取款 1000成功。
取款失败。
取款失败。
最终账号余额是:5.0
```

分析:从输出结果看,用了互斥锁后,可以保证账号状态的正确性。只有一个线程可以取款成功,其他线程因为余额不足不能取款。synchronized 关键字加在方法 withdraw()之前,表示线程执行该方法时,必须获得该方法所在对象的互斥锁,当前情况就是获得账号对象的互斥锁。一般凡是需要获得互斥锁的地方都可以使用 synchronized。对于本例,也可

以把 synchronized 用在账号类的前面,表示该类的所有方法都是同步方法。

9.5.2 线程的同步

多线程并发工作时,有的时候需要互斥,有的时候需要相互协作,按照一定的步骤共同完成任务。

以银行账号为例,有两类线程:存款线程和取款线程。存款线程负责向账号存款,取款线程负责从账号取款。操作的条件是:当账户余额为 0 时,不能取款,取款线程等待;当账户余额达到 1 万时,不能继续存款,存款线程等待。

这个问题符合生产者—消费者模型:①生产者生产商品保存到厂库;②消费者从厂库取走商品;③仓库容量有限,只有当仓库有剩余空间时,生产者可以把商品加入仓库,否则等待;④只有仓库非空时,消费者才能取走产品,否则等待。

Java 为多线程同步提供了两类方法:wait()和 notify()方法。wait()方法的语义是,当一个线程执行了该方法时,放弃互斥锁,进入互斥锁的等待队列。notify()方法的语义是,该方法唤醒互斥锁等待队列中的线程,并进入就绪状态。

【例 9.10】 多线程同步操作银行账号。

```
package code0905;
class SynAccount {
    double balance;                    //余额
    final double MAX=10000;            //最高限额
    public SynAccount(double balance) {
        this.balance=balance;
    }
//取款同步方法
    public synchronized void withdraw(double money) {
        if (balance<money) {
            try {
                System.out.printf("取款%1$,.2f失败。余额:%2$,.2f\n", money,
                balance);
                wait();            //进入等待队列
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        balance -=money;
        System.out.printf("取款%1$,.2f成功。余额:%2$,.2f\n", money,balance);
        notify();                  //唤醒等待队列的线程
    }
//存款同步方法
    public synchronized void deposit(double money) {
        if (balance+money>=MAX) {
            try {
                System.out.printf("存款%1$,.2f失败。余额:%2$,.2f\n", money,
```

```
                    balance);
                    wait();            //进入等待队列
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
            balance+=money;
            System.out.printf("存款%1$,.2f 成功。余额:%2$,.2f\n", money,balance);
            notify();                  //唤醒等待队列的线程
        }
    }
public class SynThreads {
    //取款线程
    static class Withdrawer extends Thread {
        SynAccount account;
        public Withdrawer(SynAccount account) {
            super();
            this.account=account;
        }
        @Override
        public void run() {
            for (int i=0; i<8; i++) {
                account.withdraw(2000);
            }
        }
    }
    //存款线程
    static class Depositor extends Thread {
        SynAccount account;
        public Depositor(SynAccount account) {
            this.account=account;
        }
        @Override
        public void run() {
            for (int i=0; i<8; i++) {
                account.deposit(2000);
            }
        }
    }
    public static void main(String[] args) {
        SynAccount account=new SynAccount(5000);
        Thread withdraw=new Withdrawer(account);
        Thread deposit=new Depositor(account);
        withdraw.start();
        deposit.start();
```

```
        }
    }
```

程序运行结果:

```
取款 2,000.00成功。余额:3,000.00
取款 2,000.00成功。余额:1,000.00
取款 2,000.00失败。余额:1,000.00
存款 2,000.00成功。余额:3,000.00
存款 2,000.00成功。余额:5,000.00
存款 2,000.00成功。余额:7,000.00
存款 2,000.00成功。余额:9,000.00
存款 2,000.00失败。余额:9,000.00
取款 2,000.00成功。余额:7,000.00
取款 2,000.00成功。余额:5,000.00
取款 2,000.00成功。余额:3,000.00
取款 2,000.00成功。余额:1,000.00
取款 2,000.00失败。余额:1,000.00
存款 2,000.00成功。余额:3,000.00
存款 2,000.00成功。余额:5,000.00
存款 2,000.00成功。余额:7,000.00
存款 2,000.00成功。余额:9,000.00
取款 2,000.00成功。余额:7,000.00
取款 2,000.00成功。余额:5,000.00
```

分析:从输出结果可以看出,存款线程和取款线程实现了同步协作工作。条件不满足时,存款和取款都会失败。该例使用了格式化输出,请参考 8.3.3 节。

多线程同步时,需要注意以下问题:

(1) wait()和 notify()方法必须位于同步代码块中,也就是在 synchronized 代码块中。执行这些方法的线程必须已经获得了互斥锁。这两个方法属于拥有互斥锁的对象。

(2) wait()和 notify()方法必须配对使用,执行 wait()方法进入等待队列的线程,应该由另一个线程执行 notify()方法唤醒。

(3) 在某些情况下,可以使用 notifyAll()方法代替 notify()方法,唤醒等待队列中的所有线程。

9.5.3 实用案例

【例 9.11】 使用显示锁实现多线程互斥。

包 java. util. concurrent. lock 中的类 ReentrantLock 作为显示锁,替代 Java 语言中 synchronized 的功能,在争用条件下却有更好的性能。它有一个与锁相关的获取计数器,如果拥有锁的某个线程得到锁,那么获取计数器就加 1。锁必须在 finally 块中释放。否则,如果受保护的代码将抛出异常,锁就有可能永远得不到释放。

下面用显示锁实现多线程取款操作。

程序示例代码如下:

```
package code0905;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;
public class LockAccount {
    double balance;                    //账号余额
    Lock lock=new ReentrantLock();创建锁对象
    public LockAccount(double balance) {
        super();
        this.balance=balance;
    }
    //使用显示锁
    public void withdraw(double money) {
        lock.lock();                 //锁定
        try {
            if (balance>=1000) {
                try {
                    Thread.sleep(100);
                    balance -=1000;
                    System.out.println("取款 1000 成功。");
                } catch (Exception e) {
                    e.printStackTrace();
                }
            } else {
                System.out.println("取款失败。");
            }
        } finally {
            lock.unlock();          //释放锁
        }
    }
}
class AccountThread3 extends Thread {
    LockAccount account;
    int delay=100;
    public AccountThread3(LockAccount account) {
        super();
        this.account=account;
    }
    @Override
    public void run() {
        account.withdraw(1000);
    }
    public static void main(String[] args) throws InterruptedException {
        //创建账号对象
        LockAccount account=new LockAccount(1005);
        //创建 3个取款线程
```

```
            AccountThread3 t1=new AccountThread3(account);
            AccountThread3 t2=new AccountThread3(account);
            AccountThread3 t3=new AccountThread3(account);
            t1.start();
            t2.start();
            t3.start();
            //主线程等待 3个线程结束后,输出最后余额
            t1.join();
            t2.join();
            t3.join();
            System.out.println("最终账号余额是:"+account.balance);
        }
    }
```

程序运行结果:

```
取款 1000成功。
取款失败。
取款失败。
最终账号余额是:5.0
```

分析:从输出结果可以看出,使用显示锁确保了在多线程并发操作账号的情况下数据的正确性。

9.6 多线程实训任务

【任务描述】

编写一个 Java 多线程程序,同时用 3 种方法实现对一个整型数组的排序。可以使用的排序算法有冒泡法、双向冒泡法和快速排序法。

【任务分析】

3 种排序算法并发执行,需要创建 3 个线程,每个线程执行一种排序算法。为每种算法创建一个线程目标对象。可以使用工具类 System 计算每个线程的执行时间。整型数组可以用随机数产生,并使用 System 类为线程复制数组数据。

3 种排序算法有共同的执行框架,可以定义一个父类 Sort,定义排序线程的执行框架,分别创建 BSort、QSort 和 BBSort 继承 Sort 类。分别表示冒泡排序执行体、快速排序执行体和双向冒泡排序执行体。

最后,定义一个主程序 MainTest 具体创建线程对象,执行排序算法。

【任务解决】

完整程序如下:

```
//排序算法的目标执行对象,定义排序线程执行体的框架,子类覆盖 sort 方法
public class Sort implements Runnable {
    int[] data;                                    //排序算法需要的数据
    public Sort(int[] data) {
```

```
        super();
        this.data=data;
    }
    //具体的排序资料覆盖此方法,实现排序算法
    public void sort() {
        throw new RuntimeException("please do it in subclass");
    }
    @Override
    public void run() {                    //线程体,执行 sort 方法,并输出排序后的结果
        long start=System.nanoTime();      //开始时间
        sort();                            //执行排序
        long end=System.nanoTime();        //结束时间
        System.out.println(Thread.currentThread().getName()
                    +"执行时间:"+(end-start)+"ns");
    }
}
//双向冒泡排序算法
public class BBSort extends Sort {

    public BBSort(int[] data) {
        super(data);
    }
    public void sort() {
        try {
            sort(data);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //双向冒泡排序程序,可以参考相关文献
    void sort(int a[]) throws Exception {
        int j;
        int limit=a.length;
        int st=-1;
        while(st<limit) {
            st++;
            limit--;
            boolean swapped=false;
            for (j=st; j<limit; j++) {

                if (a[j]>a[j+1]) {
                    int T=a[j];
                    a[j]=a[j+1];
                    a[j+1]=T;
                    swapped=true;
```

```
                    }
                }
                if(!swapped) {
                    return;
                } else
                    swapped=false;
                for (j=limit; --j>=st;) {
                    if (a[j]>a[j+1]) {
                        int T=a[j];
                        a[j]=a[j+1];
                        a[j+1]=T;
                        swapped=true;
                    }
                }
                if(!swapped) {
                    return;
                }
            }
        }
    }
    //冒泡排序程序
    public class BSort extends Sort {
        public BSort(int[] data) {
            super(data);
        }
        public void sort() {
            try {
                sort(data);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
        void sort(int a[]) throws Exception {
            for (int i=a.length; --i>=0;) {
                boolean swapped=false;
                for (int j=0; j<i; j++) {
                    if (a[j]>a[j+1]) {
                        int T=a[j];
                        a[j]=a[j+1];
                        a[j+1]=T;
                        swapped=true;
                    }
                }
                if(!swapped)
                    return;
```

```
        }
    }
}
//快速排序算法
public class QSort extends Sort {
    public QSort(int[] data) {
        super(data);
    }
    public void sort() {
        try {
            QuickSort(data, 0, data.length-1);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private void swap(int a[], int i, int j) {
        int T;
        T=a[i];
        a[i]=a[j];
        a[j]=T;
    }
    void QuickSort(int a[], int lo0, int hi0) throws Exception {
        int lo=lo0;
        int hi=hi0;
        int mid;
        if (hi0>lo0) {
            mid=a[(lo0+hi0)/2];
            while(lo<=hi) {
                while((lo<hi0) && (a[lo]<mid))
                   ++lo;
                while((hi>lo0) && (a[hi]>mid))
                    --hi;
                if (lo<=hi) {
                    swap(a, lo, hi);
                    ++lo;
                    --hi;
                }
            }
            if (lo0<hi)
                QuickSort(a, lo0, hi);
            if (lo<hi0)
                QuickSort(a, lo, hi0);
        }
    }
}
```

```
//主程序创建线程,执行 3 种排序算法
public class MainTest {
    public static void main(String[] args) {
        final int count=10000;
        int[] data=createData(count);        //创建排序数据
        //创建快速排序线程
        Thread qsort=new Thread(new QSort(data), "快速排序");
        int[] data2=new int[count];
        //复制需要排序的数组,为下一个排序算法准备数据
        System.arraycopy(data, 0, data2, 0, count);
        //创建冒泡排序线程
        Thread bsort=new Thread(new BSort(data2), "冒泡排序");
        int[] data3=new int[count];
        //复制需要排序的数组,为下一个排序算法准备数据
        System.arraycopy(data, 0, data3, 0, count);
        //创建双向冒泡排序线程
        Thread bbsort=new Thread(new BBSort(data3), "双向冒泡排序");
        qsort.start();
        bsort.start();
        bbsort.start();
    }
    //随机产生排序的数据
    private static int[] createData(int count) {
        int[] data=new int[count];
        for (int i=0; i<data.length; i++) {
            data[i]=(int) (Math.random() * count);
        }
        return data;
    }
}
```

程序运行结果:

```
快速排序执行时间:2258667ns
双向冒泡排序执行时间:250393276ns
冒泡排序执行时间:392050665ns
```

可以看出,在对 10000 个数据排序的情况下,快速排序算法最快,其次是双向冒泡算法,最慢是冒泡排序算法。

习题与思考

1. 将窗口分为上下两个区,分别运行两个线程,一个在上面的区域中显示由右向左游动的字符串,另一个在下面的区域中显示从左向右游动的字符串。

2. 简述程序、进程和线程之间的关系。什么是多线程程序?

3. 线程有哪5个基本状态?它们之间如何转化?简述线程的生命周期。

4. Runnable接口中包括哪些抽象方法?Thread类有哪些主要的成员变量和方法?

5. 如何在Java程序中实现多线程?简述使用Thread子类和实现Runnable接口两种方法的异同。

6. 使用非固定线程池创建线程,并执行任务。比较和固定线程池的区别。

7. 使用多线程同步和互斥方法解决5个哲学家问题。

第10章 GUI程序设计

本章学习目标

图形用户界面(Graphic User Interface,GUI)设计是构建可视化应用和Java Applet的重要基础。Java有很多GUI程序设计技术,例如AWT、Swing和SWT等。本章通过对Swing技术的介绍,使读者初步掌握GUI程序设计的基本方法,并将其应用于Java小应用程序Applet的可视化设计。本章需要重点掌握以下内容:

(1) Swing技术及其常用组件。

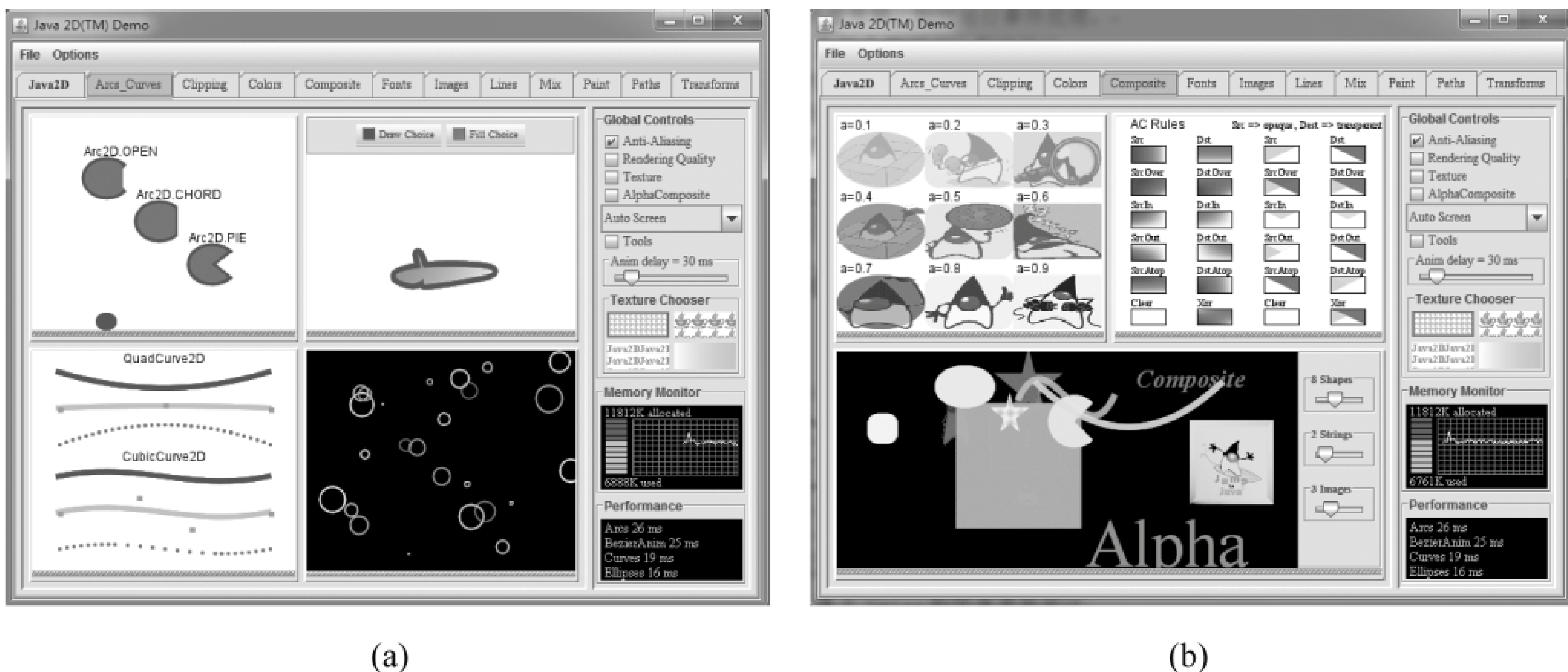
(2) 基于Swing的界面布局。

(3) GUI中的事件处理机制。

(4) 高级Swing组件JTree和JTable。

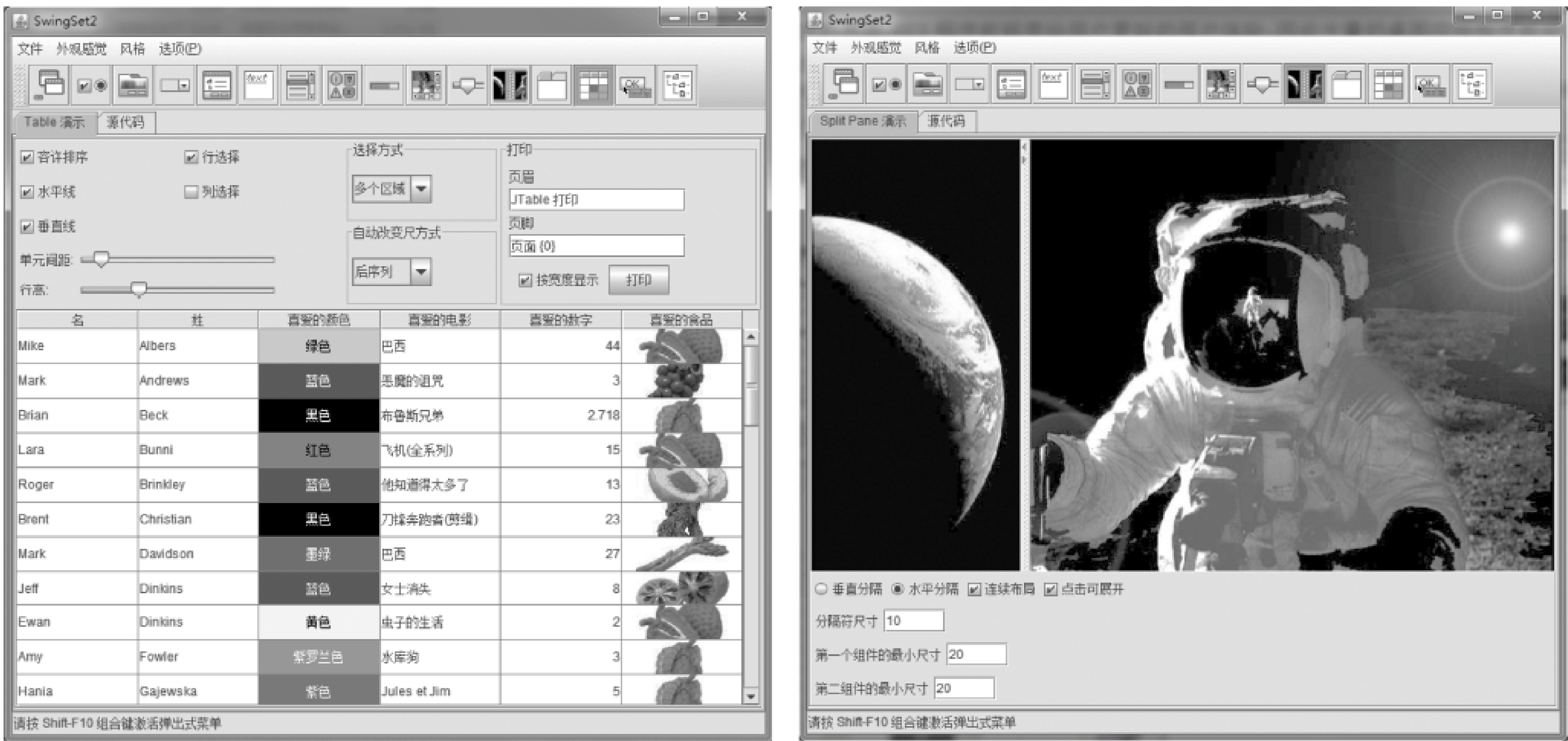
10.1 为什么学习GUI程序设计

对于程序的使用者而言,他们更愿意使用那些界面友好的应用程序。相对于命令行程序不友好的界面,GUI程序能够带给用户更好的用户体验,因此大量的桌面应用程序都会以GUI的形式提供给最终用户。图10-1所示的GUI程序都来自于JDK提供的样例程序,非常漂亮,相信通过本章的学习,读者也能开发出同样漂亮的软件界面。



(a) (b)

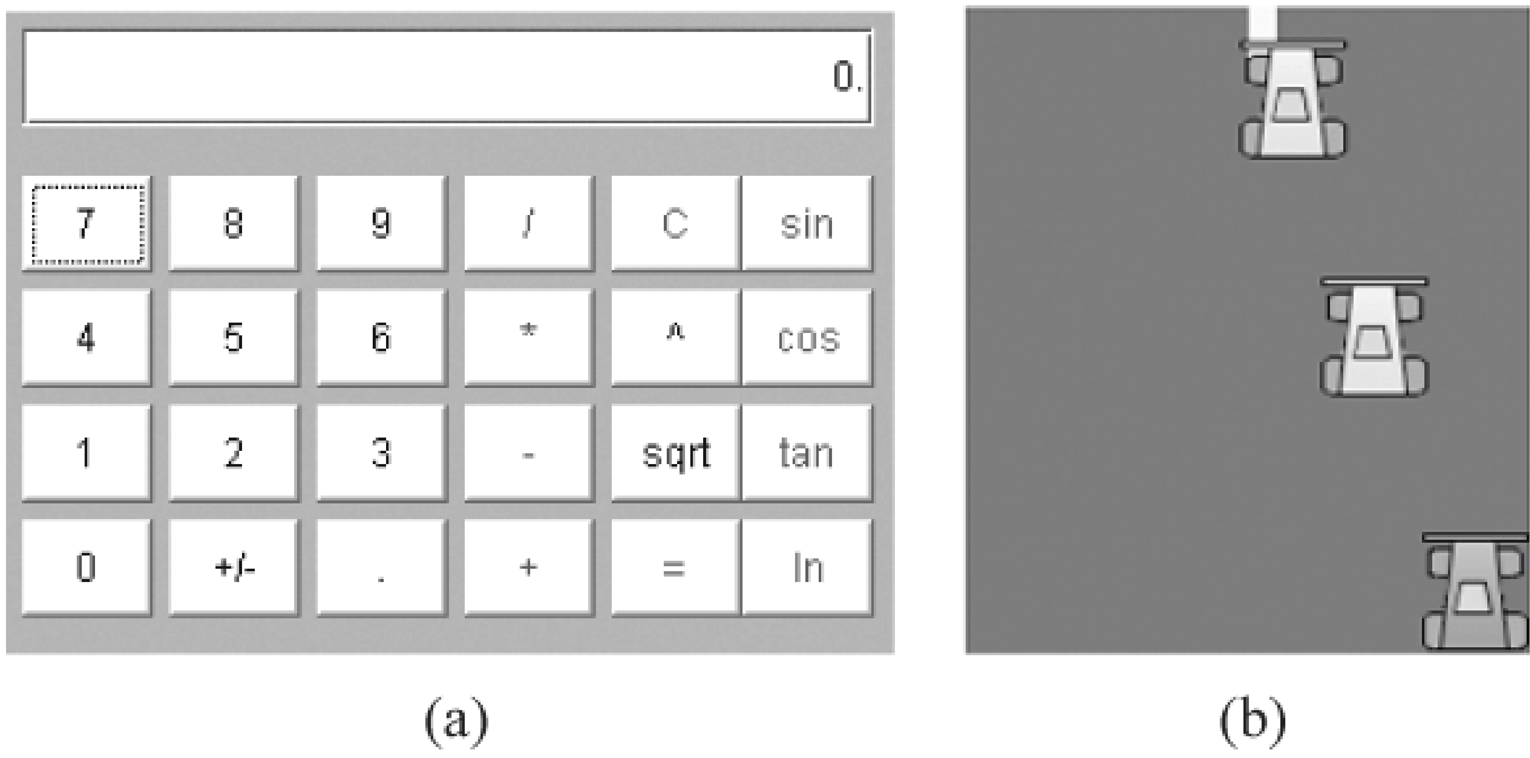
图10-1 基于Swing的GUI程序



(c) (d)

图 10-1 （续）

当然，GUI 程序带给用户的不仅是一种更直观的界面，它也提供了用户一种更友好的交互方式，用户可以通过鼠标单击、拖动、键盘控制等更灵活的方式进行应用操作。例如，基于鼠标单击的“运算器”，基于键盘←、→的“赛车游戏”等，如图 10-2 所示。同样，通过学习 10.4 节中 Java 事件处理机制，读者也能自己开发出类似具有交互功能的应用。



(a) (b)

图 10-2 具有交互能力的 GUI 程序

10.2 基于 Swing 的简单界面设计

10.2.1 Swing 简介

Java 的 GUI 程序设计技术主要包括 AWT(Abstract Window Toolkit)、Swing 和 SWT (Standard Widget Toolkit)。其中，SWT 是 IBM 公司最早提出的开源 GUI 程序设计 API，使用 SWT 需要从网上下载安装额外的 Java 包。AWT 和 Swing 是 Java SE 自带的标准 GUI 程序设计 API，因此使用 AWT 或者 Swing 编写 GUI 程序，只需要正确安装 Java SE 就可以了。

从 Java 1.0(JDK 1.0)发布开始，AWT 就是 JDK 的一部分，当时还没有 Swing 技术。随着开发人员将 Java 应用于越来越多的平台，AWT 的弱点开始逐渐暴露。其中，AWT 最

主要的问题是:AWT 只提供了建立窗口操作应用程序所必需的最少功能,对于构建复杂的窗体程序(类似于 Word、PowerPoint 一样的程序),AWT 提供的功能是远远不足的。

Swing 是在 AWT 基础上发展起来的一项技术。Swing 不仅包括了 AWT 所具有的全部组件,而且可以使用树形组件(JTree)、表格(JTable)、选项卡(JTabbedPane)等高级图形组件。同时,Swing 完全由 Java 编写,不再依赖于运行时平台的本地组件(AWT 和 SWT 都存在本地调用),具有良好的可移植性。

由于 Swing 不仅包含了 AWT 的全部功能,而且具有更多高级的特性,随着 Java 技术的发展,用 Swing 替代 AWT 已经成为一种趋势。事实上 AWT 和 Swing 技术极为相似,GUI 组件的类名通常只是比 AWT 组件的类名多一个字 J。例如,按键组件在 Swing 中对应 JButton 类,而在 AWT 中对应 Button 类。因此,掌握了 Swing,学习 AWT 就非常容易了。

10.2.2 Swing 的类层次结构

Swing 由很多 Java 类构成。这些 Java 类主要包含在 javax.swing 这个 Java 包中。图 10-3 是 Swing 的类层次结构,其中的 Java 类可以分为以下两大类。



图 10-3 Swing 的类层次结构

1. JComponent 及其子类

JComponent 及其子类,简称 GUI 组件。GUI 组件中有一部分组件具有图形外观,能

在图形界面上与用户进行交互，称为可视化组件。例如，JButton、JLabel、JTextField 等就是可视化组件。GUI 组件中的另外一些组件没有图形外观，称为非可视化组件。非可视化组件通常需要与可视化组件相结合，共同完成特定的图形功能。例如，JPanel 就是典型的非可视化组件，主要用于界面的布局。

2. 顶层容器

顶层容器(container)包含 JApplet、JDialog、JFrame 和 JWindow 及其子类。通常将 GUI 组件布局在顶层容器上来实现 GUI 程序的界面布局。

JFrame 是最常用的一种顶层容器，它的作用是创建一个顶层的 Windows 窗体。JFrame 的外观就像平常 Windows 系统下见到的窗体，有标题、边框、菜单等。

JDialog 用于创建对话框。JDialog 创建的对话框相对于 JFrame 创建的窗体没有最大化和最小化按键。在 GUI 编程时可以根据需要选择使用 JDialog 还是 JFrame。

JWindow 创建的窗体没有标题栏，没有最大化和最小化按键。在某些 GUI 应用中，可能需要编写这种不带修饰的窗体，或者用户希望用自己编写的标题栏、最大化、最小化按键来替换 Windows 自带的窗体风格，此时就可以选择创建 JWindow 来实现这些窗体效果。

Applet 是一种能够嵌入到网页中执行的 Java 图形程序。JApplet 是创建这种程序的顶层容器。除了顶层容器，Swing 中还有一些轻量级的容器，例如 JPanel。通常可以先将 GUI 组件放置在 JPanel 上，再将 JPanel 加入其他容器或者顶层中，从而实现复杂的界面布局。

10.2.3 常见 GUI 组件

Swing 包含了非常多的 GUI 组件，其中常见的组件包括 JButton(按键)、JLabel(标签)、JTextField(单行文本框)和 JTextArea(文本输入区)、JTree(树状组件)、Table(表格)等。

1. 按键 JButton

按键是最常用的一个组件，其相应的类是 JButton，它的常用构造方法如下：

(1) JButton()：构造一个没有文字、没有图片的按键。

(2) JButton(Icon icon)：构造一个带图片的按键，icon 是按键上的图片。

(3) JButton(String text)：构造一个带文字的按键，text 是按键上的文字。

(4) JButton(String text, Icon icon)：构造一个带文字、带图片的按键，text 和 icon 分别对应按键所使用的文字和图片。

2. 标签 JLabel

JLabel 组件用于显示标签，它的常用构造方法如下：

(1) JLabel(String text)：构造一个文字标签，text 是标签对应的文字。

(2) JLabel(Icon image)：构造一个图片标签，image 是标签对应的图像。

(3) JLabel(String text, int horizontalAlignment)：构造一个文本标签，并通过 horizontalAlignment 指定文本对齐方式，horizontalAlignment 的可取值包括 JLabel.LEFT、JLabel.RIGHT 等。

(4) JLabel(String text,Icon icon,int horizontalAlignment)：构造一个带文本、带图片的标签，horizontalAlignment 指定对齐方式。

3. 单行文本框 JTextField

JTextField 用于接受用户的输入信息,但它只能接受一行的用户输入信息,它常用构造方法如下:

(1) JTextField():构造一个空的单行文本框。

(2) JTextField(int columns):构造一个空的单行文本框,文本框的宽度(列数)为 columns。

(3) JTextField(String text):构造一个单行文本框,文本框的初始内容为 text。

(4) JTextField(String text, int columns):构造一个单行文本框,文本框的初始内容为 text,宽度(列数)为 columns。

4. 文本输入区 JTextArea

JTextArea 也称为多行文本框,与 JTextField 不同,JTextArea 可以显示多行多列的文本,其常用构造方法如下:

(1) JTextArea():创建一个空的文本输入区。

(2) JTextArea(int rows, int columns):创建一个 rows 行、columns 列的文本输入区。

(3) JTextArea(String text):创建一个文本输入区,文本输入区的初始化文本为 text。

(4) JTextArea(String text, int rows, int columns) 创建一个 rows 行、columns 列的文本输入区,文本输入区的初始化文本为 text。

【例 10.1】 使用 Swing JButton 的 GUI 程序。

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
public class SwingFrame extends JFrame{
    private JButton button= new JButton("按键");
    public SwingFrame()
    {
        setSize(300,300);                      //设置窗体大小
        setLocation(400, 400);                 //设置窗体显示位置
        setTitle("ButtonFrame");               //设置窗体标题栏
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);           //设置窗体默认关闭事件
        setLayout(new FlowLayout());           //设置布局管理器
        //添加按键事件
        button.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent event) {
                JOptionPane.showMessageDialog(null, "单击了按键!");
            }
        });
        add(button);                           //添加按键
    }
    public static void main(String[] args) {
```

```
        SwingFrame frame=new SwingFrame();
        frame.setVisible(true);                //显示窗体
    }
}
```

使用上述 Swing JButton 的 GUI 程序的显示效果如图 10-4 所示。



图 10-4　使用 Swing JButton 的 GUI 程序的显示效果

分析：例 10.1 通过从 JFrame 继承，创建了 ButtonFrame 类。在 ButtonFrame 类的构造函数里，调用 setSize 方法设置窗体大小，setLocation 方法设置窗体显示的位置，setTitle 方法设置窗体的标题。setDefaultCloseOperation 方法用于设置窗体的默认关闭操作，JFrame.EXIT_ON_CLOSE 表明整个应用程序都终止运行。

此外，还有 JFrame.DISPOSE_ON_CLOSE、JFrame.DO_NOTHING_ON_CLOSE 和 JFrame.HIDE_ON_CLOSE 等参数。如果不用 setDefaultCloseOperation 方法设置默认关闭操作，则单击窗体的关闭按键时程序不会终止而只会隐藏窗体。button.addActionListener 为按键添加事件响应。setLayout 方法用于设置窗体的布局管理器。最后通过调用 add 方法将一个 JButton 对象添加到窗体上。程序的 main 函数首先创建 ButtonFrame 类的实例，然后调用 setVisible 方法显示窗体。

例 10.1 所示代码给出了 GUI 程序的基本流程：

(1) 根据需要从相应的顶层容器继承(如果创建窗体就继承 JFrame，对话框就继承 JDialog)，新建一个子类。

(2) 然后设置顶层容器的属性，包括大小、位置、标题和关闭事件等。

(3) 设置界面上 GUI 组件的事件响应。

(4) 向顶层容器上添加 GUI 组件，并进行布局。

(5) 创建新建子类的实例，调用 setVisible 方法显示界面。

10.2.4　基于 AWT 的 GUI 程序

看过了 Swing 编写的应用程序，读者一定会好奇 AWT 编写的 GUI 程序又是怎样的一种形式呢？实际上 AWT 编写的 GUI 程序与 Swing 有非常相似的地方。例 10.2 给出了一个用 AWT 编写的应用程序。它与前几个程序的不同之处在于：

(1) JFrame 和 JButton 替代为 Frame 和 Button。

(2) Frame 类没有 setDefaultCloseOperation 方法,因此用 addWindowListener 方法来添加窗口关闭事件。

(3) JOptionPane 类是一个 Swing 类,但由于 AWT 没有类似的类,因此在例 10.2 中用到了这个类。从中可以看出 Swing 和 AWT 类是可以混用的。

可以看出,AWT 程序与 Swing 程序基本类似,但 Swing 在编写 GUI 程序时比 AWT 更为方便。

【例 10.2】 使用 AWT 的 GUI 程序。

```
import java.awt.Button;
import java.awt.FlowLayout;
import java.awt.Frame;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import javax.swing.JOptionPane;
public class AWTFrame extends Frame{
    private Button button=new Button("按键");
    public AWTFrame()
    {
        setSize(300,300);                   //设置窗体大小
        setLocation(400, 400);              //设置窗体显示位置
        setTitle("AWTFrame");               //设置窗体标题栏
        //设置窗体关闭事件
        this.addWindowListener(new WindowAdapter(){
            public void windowClosing(WindowEvent arg0) {
                System.exit(0);
            }
        });
        setLayout(new FlowLayout());        //设置布局管理器
        //添加按键事件
        button.addActionListener(new ActionListener(){
            public void actionPerformed(ActionEvent event) {
                JOptionPane.showMessageDialog(null, "单击了按键!");
            }
        });
        add(button);                        //添加按键
    }
    public static void main(String[] args) {
        AWTFrame frame=new AWTFrame();
        frame.setVisible(true);             //显示窗体
    }
}
```

使用上述 AWT 的 GUI 程序的显示效果如图 10-5 所示。



图 10-5　使用 AWT 的 GUI 程序的显示效果

10.3　界面布局

创建 GUI 程序最重要的是界面布局。Swing 采用了两种布局方式：无布局管理器布局和基于布局管理器的布局。其中，无布局管理器布局类似于 Visual Basic、Delphi 和 Visual C++ 采用的布局方式，通过指定 GUI 组件在窗体上的绝对位置进行组件布局。

基于布局管理器的布局，首先通过调用容器类（JFrame、JDialog 或 JPanel 等）的 setLayout 方法设置布局管理器（包括 FlowLayout、BorderLayout、GridLayout 等）。设置好布局管理器之后，容器内的所有组件的布局就由布局管理器负责，包括组件的排列顺序、组件的大小和位置，以及当窗口移动或调整大小后组件如何变化等。

采用布局管理器比无布局管理方式更具有灵活性，例如当窗体的大小或者分辨率发生改变时，采用布局管理器方式能够重新布局组件，而采用无布局管理器布局则需要编程者去控制新的组件的位置和大小。同时，采用布局管理器的 Swing 程序跨平台的布局效果比采用无布局管理器布局更好。

10.3.1　无布局管理器布局

采用无布局管理器布局，首先要取消 Swing 默认的布局管理器，否则布局方法不会生效。其方法为调用容器的 setLayout 方法，并将布局管理器设置为 null。取消默认布局管理器之后，就可以使用 GUI 组件的 setLocation()、setSize()、setBounds() 等布局方法来对 GUI 组件的位置、大小进行设置。表 10-1 列出了 JComponent 及其子类所包含的常用的布局方法。

表 10-1　常用的布局方法

函　　数	作　　用
setLocation(java.awt.Point) setLocation(int,int)	设置组件的坐标位置
setSize(java.awt.Dimension) setSize(int,int)	设置组件的大小

续表

函　　数	作　　用
setBounds(java. awt. Rectangle) setBounds(int, int, int, int)	同时设置组件的坐标位置和大小。setBounds(int, int, int, int)的 4 个参数分别是 x、y、width、height，即组件的(x, y)坐标，以及组件的 width 和 height

【例 10.3】 无布局管理器的示例。

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JTextField;
public class AbsoluteLayoutDemo extends JFrame {
    private JButton button=new JButton("按键");
    private JTextField textField=new JTextField("文本框");
    public AbsoluteLayoutDemo()
    {
        setSize(500, 200);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //设置布局管理器为 null
        setLayout(null);
        //设置输入框的位置为 (20,20),宽 200,高 100
        textField.setBounds(20,20,200,100);
        add(textField);
        //设置按键的位置为 (300,50),宽 100,高 20
        button.setLocation(300, 50);
        button.setSize(100, 20);
        add(button);
    }
    public static void main(String[] args) {
        AbsoluteLayoutDemo frame=new AbsoluteLayoutDemo();
        frame.setVisible(true);
    }
}
```

例 10.3 的运行结果如图 10-6 所示。

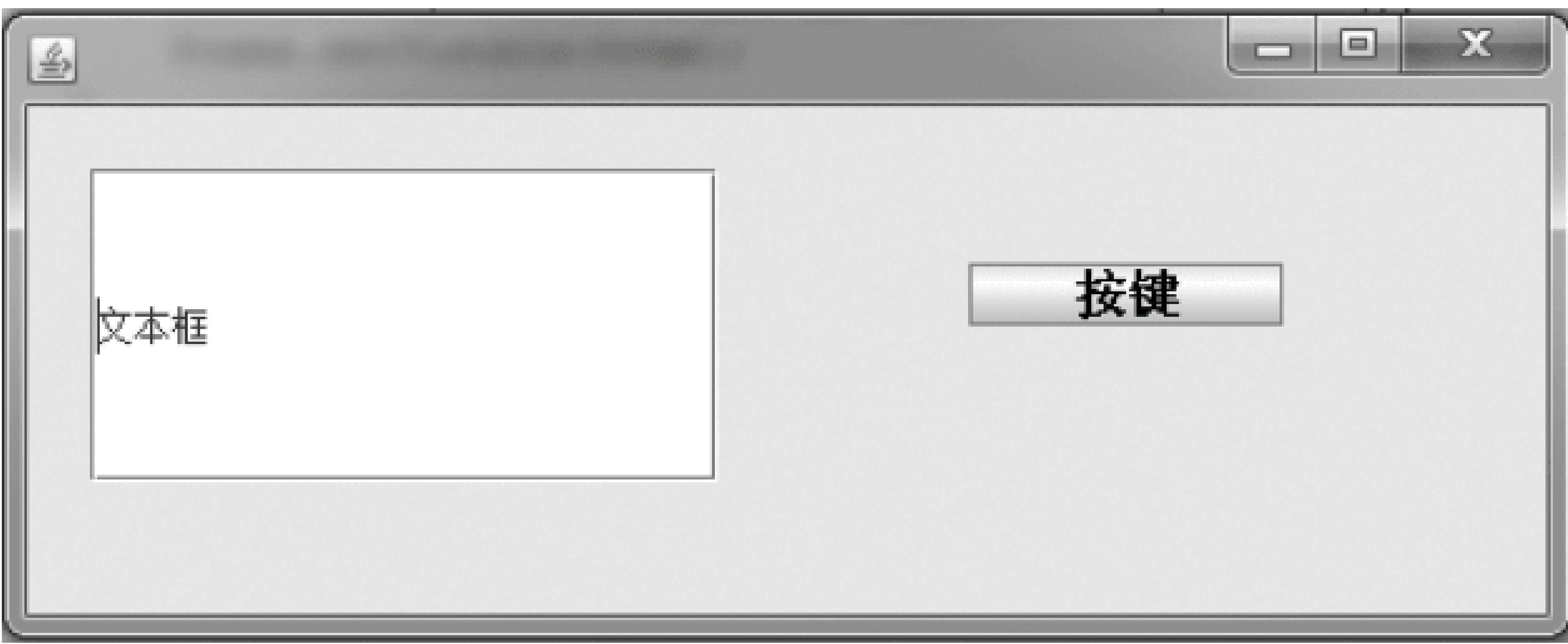


图 10-6　无布局管理器布局

无布局管理器布局在对组件的大小和位置的控制上较为灵活，但这种布局方式会导致平台相关，在不同的平台上可能产生不同的显示效果。并且在窗体发生变化时，有可能需要进行重新布局。基于布局管理器的布局可以很好地解决以上两个问题。

10.3.2 FlowLayout

界面采用 FlowLayout 布局，其 GUI 组件的放置规律是从左到右、从上到下进行放置。如果界面足够宽，第一个组件先添加到容器中第一行的最左边，后续的组件依次添加到上一个组件的右边，如果当前行已放置不下该组件，则放置到下一行的最左边。

当容器的大小发生变化时，用 FlowLayout 管理的组件会发生变化，其变化规律是：组件的大小不变，但是相对位置会发生变化。

【例 10.4】 使用 FlowLayout 的示例。

```
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class FlowLayoutDemo extends JFrame {
    private JButton button1=new JButton("First Button");
    private JButton button2=new JButton("Second Button");
    private JButton button3=new JButton("Third Button");
    private JButton button4=new JButton("Fourth Button");
    public FlowLayoutDemo() {
        setSize(300, 150);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //设置布局方式为 FlowLayout
        setLayout(new FlowLayout());
        //添加按键,注意设置布局方式之后任何对组件进行设置的方法,例如 setSize、
        //setLocation 等都会失效
        add(button1);
        add(button2);
        add(button3);
        add(button4);
    }
    public static void main(String arg[]) {
        FlowLayoutDemo frame=new FlowLayoutDemo();
        frame.setVisible(true);
    }
}
```

FlowLayout 布局的运行结果如图 10-7 所示。图 10-7(a)是程序默认运行的情况，图 10-7(b)是改变窗口大小之后的界面情况。可以看出当窗体的大小发生改变时，程序可以自行调整界面元素的位置布局，而不需要额外编写程序代码。



(a) (b)

图 10-7　FlowLayout 布局

10.3.3　BorderLayout

BorderLayout 布局把界面分成 5 个区域：North、South、East、West 和 Center，每个区域只能放置一个组件。当界面的大小发生变化时，其变化规律为：组件的相对位置不变，大小发生变化。例如，容器变高了，则 North、South 区域不变，而 West、Center、East 区域变高；如果容器变宽了，而 West、East 区域不变，而 North、Center、South 区域变宽。由于不一定所有的区域都有组件，如果四周的区域(West、East、North、South 区域)没有组件，则由 Center 区域去补充。

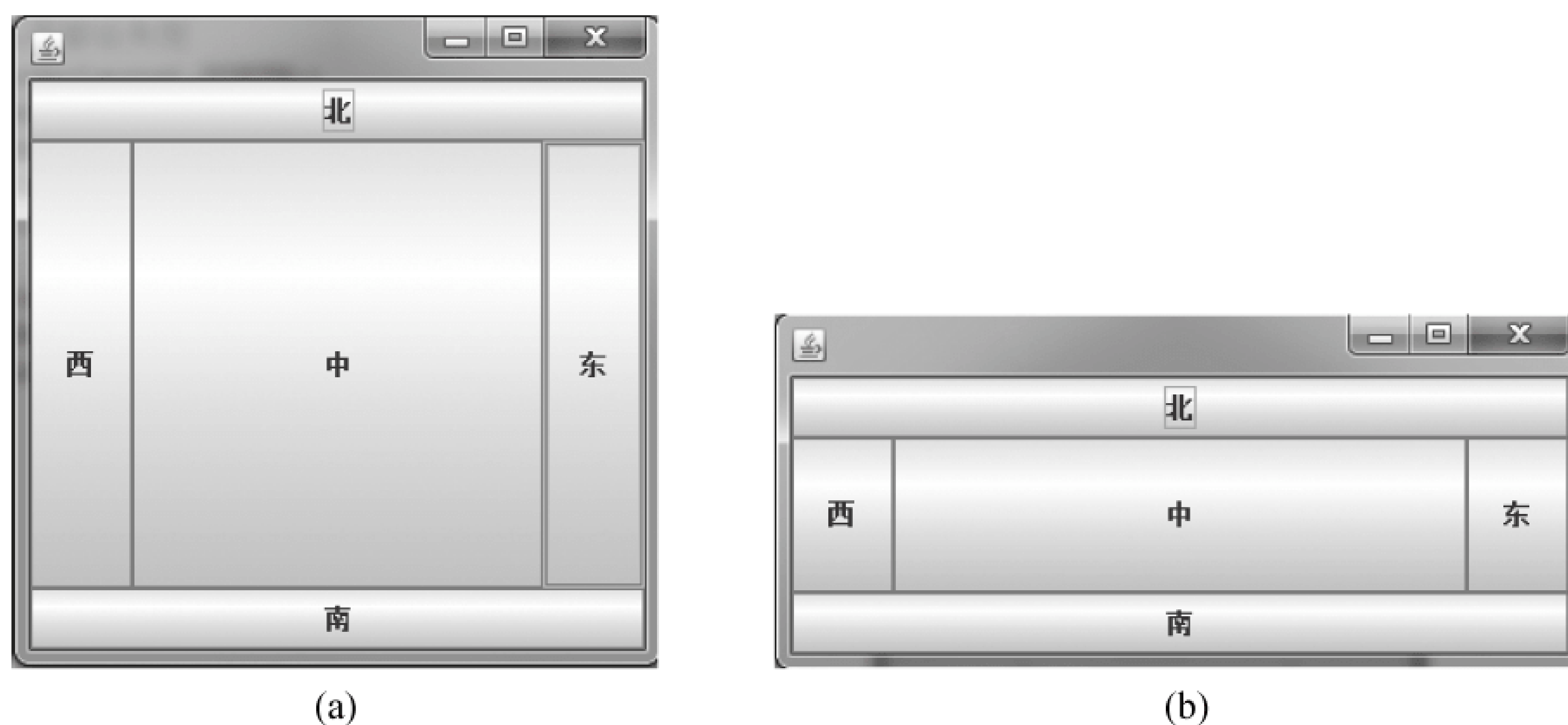
下面是 BorderLayout 布局的示例。图 10-8(a)是程序默认的界面状态，图 10-8(b)是界面大小发生变化之后的状态。

【例 10.5】 使用 BorderLayout 的示例。

```
import java.awt.BorderLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class BorderLayoutDemo extends JFrame {
    private JButton north=new JButton("北");
    private JButton south=new JButton("南");
    private JButton east=new JButton("东");
    private JButton west=new JButton("西");
    private JButton center=new JButton("中");
    public BorderLayoutDemo() {
        setSize(300, 300);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //设置布局方式为 BorderLayout
        setLayout(new BorderLayout());
        //添加按键,注意设置布局方式之后任何对组件进行设置的方法,例如 setSize、
        //setLocation 等都会失效
        add(north,BorderLayout.NORTH);
        add(south,BorderLayout.SOUTH);
        add(east,BorderLayout.EAST);
        add(west,BorderLayout.WEST);
        add(center,BorderLayout.CENTER);
    }
```

```
    public static void main(String arg[]) {
        BorderLayoutDemo frame=new BorderLayoutDemo();
        frame.setVisible(true);
    }
}
```

程序运行结果如图 10-8 所示。



(a) (b)

图 10-8 BorderLayout 布局

10.3.4 GridLayout

GridLayout 布局管理器将整个界面划分成 N 行、M 列的网格，每个网格的大小相同。布局时，按照组件加入的顺序优先考虑按行布局，当一行布满之后再布局下一行（每行只能布局 M 个组件）。只有当行列不能满足指定的数值时（N×M 小于组件个数），才按照行进行扩展。例如，5 个按钮，指定分 2 行 2 列显示，由于 2×2 只能满足 4 个按键，因此自动扩展为 3 列。

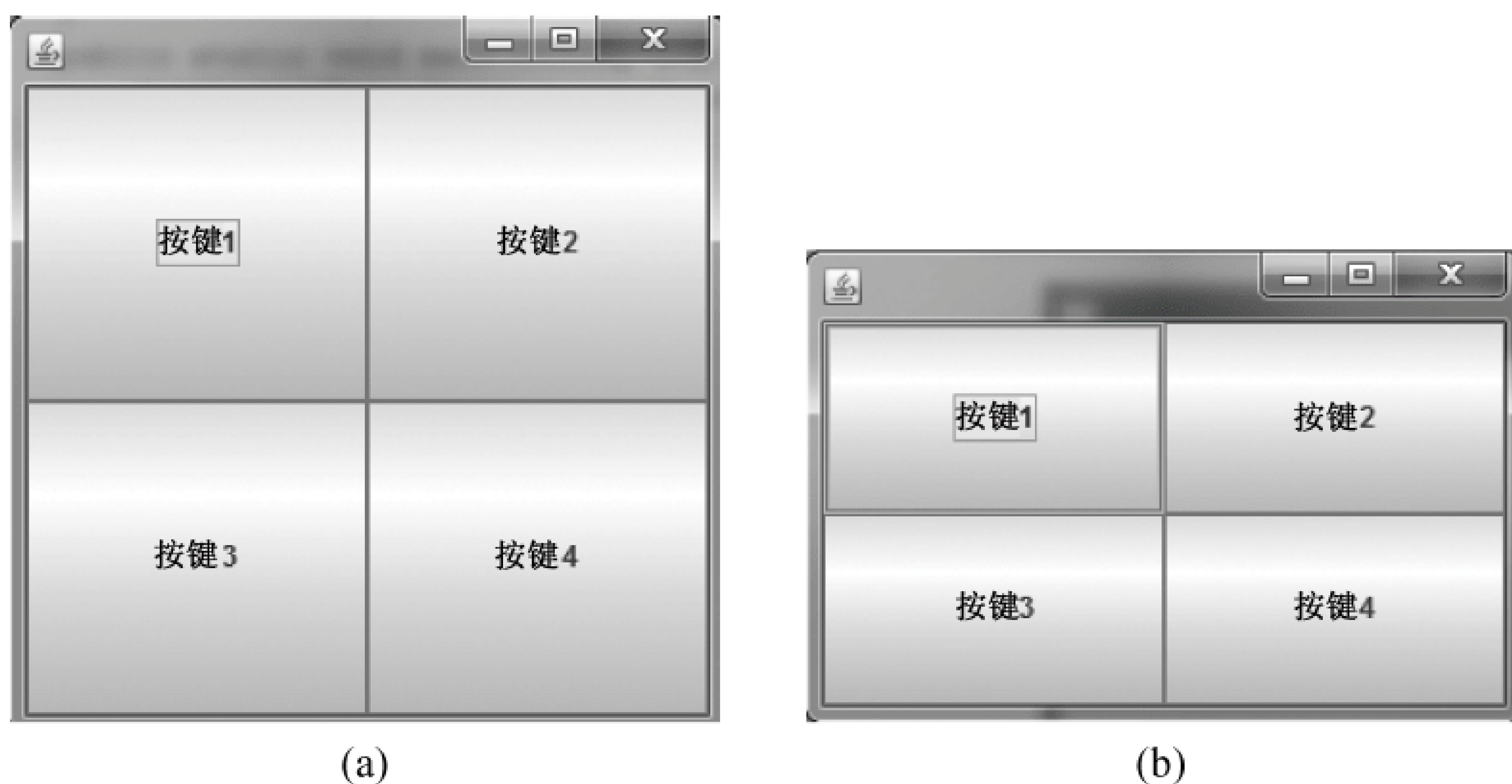
下面是 GridLayout 布局的示例。图 10-8(a)是程序默认的界面状态，图 10-8(b)是界面大小发生变化之后的状态。

【例 10.6】 使用 GridLayout 的示例。

```
import java.awt.GridLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
public class GridlayoutDemo extends JFrame {
    private JButton button1=new JButton("按键 1");
    private JButton button2=new JButton("按键 2");
    private JButton button3=new JButton("按键 3");
    private JButton button4=new JButton("按键 4");
    public GridlayoutDemo() {
        setSize(300, 300);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //设置布局方式为 GridLayout,2行、2列
```

```
            setLayout(new GridLayout(2,2));
            //添加组件时不需要设置组件所在行、列
            add(button1);
            add(button2);
            add(button3);
            add(button4);
        }
        public static void main(String arg[]) {
            GridlayoutDemo frame=new GridlayoutDemo();
            frame.setVisible(true);
        }
    }
```

程序运行结果如图 10-9 所示。



(a) (b)

图 10-9 GridLayout 布局

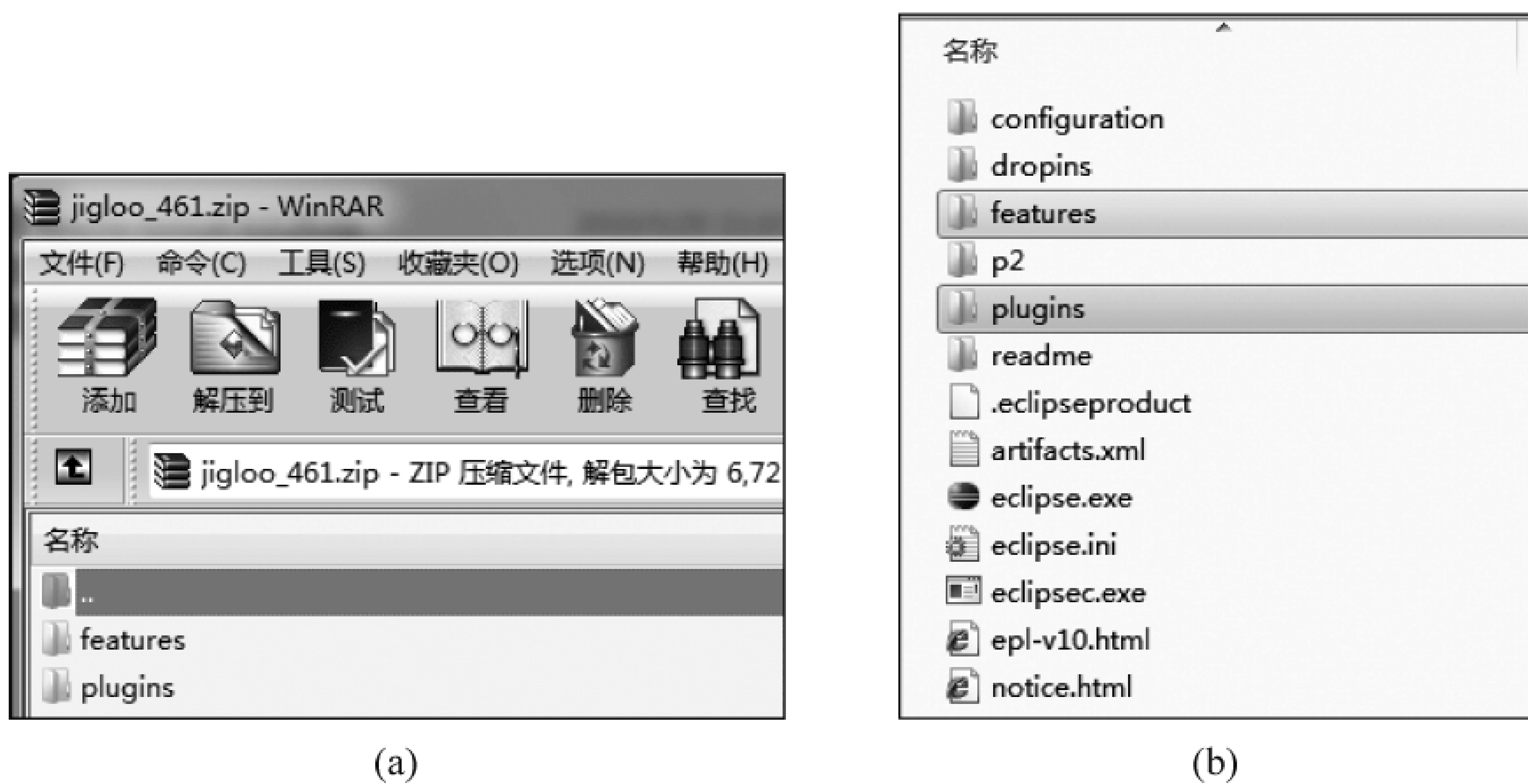
10.3.5 利用可视化工具进行布局

在 Swing 发展的早期,对界面的布局只能用编写代码的方式完成,因而是一件非常复杂的事情。但是,随着 Java 开发工具的发展,目前多数集成开发工具都提供了相当好用的可视化布局工具。例如,Eclipse 可以通过安装 Jigloo 插件来获得可视化布局工具。Jigloo 是用于辅助图形界面程序设计的插件。在非商用开发中可以免费使用 Jigloo 插件,而在商用软件开发中使用 Jigloo 插件就需要支付费用。下面简要介绍如何用 Jigloo 进行可视化布局。

首先在 Eclipse 中安装好 Jigloo 插件。安装 Jigloo 插件的第一步是从 Jigloo 官方网站(http://www.cloudgarden.com/jigloo/)下载 Jigloo 插件。Jigloo 插件是一个压缩包(本书下载的是 jigloo_461.zip 文件)。该压缩文件包含 features 和 plugins 两个目录,如图 10-10(a)所示。将这两个目录解压到 Eclipse 的安装目录(本书为 E:\ eclipse),如图 10-10(b)所示。

注意:Eclipse 的安装目录下本来就有 features 和 plugins 两个目录,解压 Jigloo 插件时直接覆盖两个目录,即可完成 Jigloo 插件的安装。Eclipse 其他插件的安装与 Jigloo 大同小异,如果在安装其他插件时遇到问题,可以参考该插件的安装说明,这些安装说明通常在插

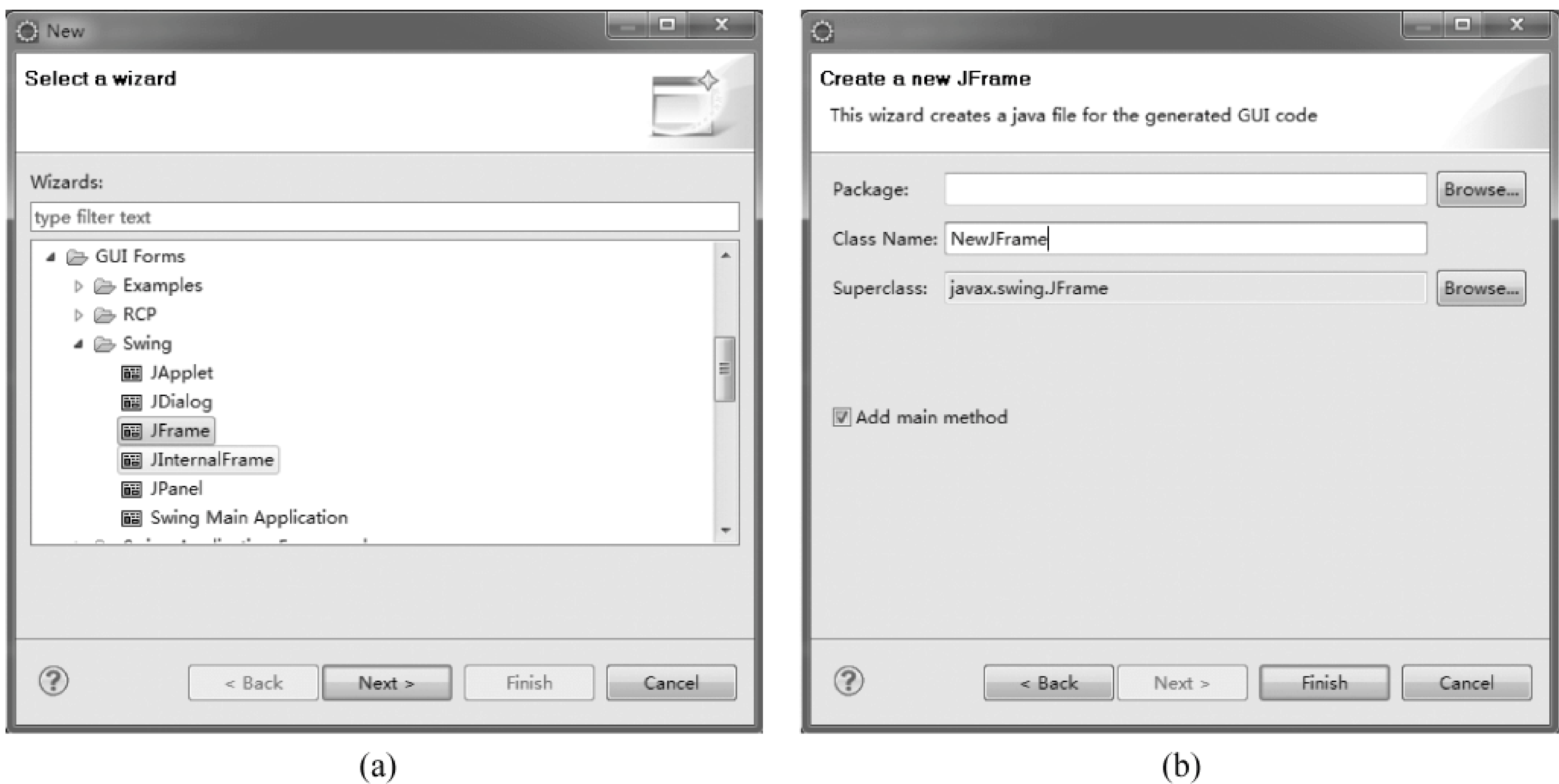
件的官方网站上都可以找到。



(a) (b)

图 10-10 安装 Jigloo 插件

插件安装完毕之后,新建一个 Java 工程(本书为“GUI 程序设计”)。选中该工程,打开右键菜单,选中菜单项 New,选择选项 other。在弹出的对话框中选择 GUI Forms→Swing→JFrame,如图 10-11(a)所示,最后按 Next 按钮。在弹出的对话框中输入包名和类名(本书建立的类为 NewJFrame),按 Finish 按钮,新建一个窗体,如图 10-11(b)所示。



(a) (b)

图 10-11 新建窗体

新建窗体之后,就可以在如图 10-12 所示的界面对窗体进行可视化编辑。该界面的左上部是界面的可视化编辑区,左下部是自动生成的程序代码,右下部是可以编辑当前选中的窗体或者 GUI 组件的属性。

下面演示一下如何进行界面布局。首先选中窗体,打开右键菜单,选择 Set Layout 为窗体设置一个布局管理器,如图 10-13(a)所示。可以看到 Swing 除了 FlowLayout、BorderLayout 和 GridLayout 之外,还有许多其他的布局管理器,这里选择 GroupLayout。GoupLayout 以 Group(组)为单位来管理布局,也就是把多个组件(如 JLable、JButton)按区

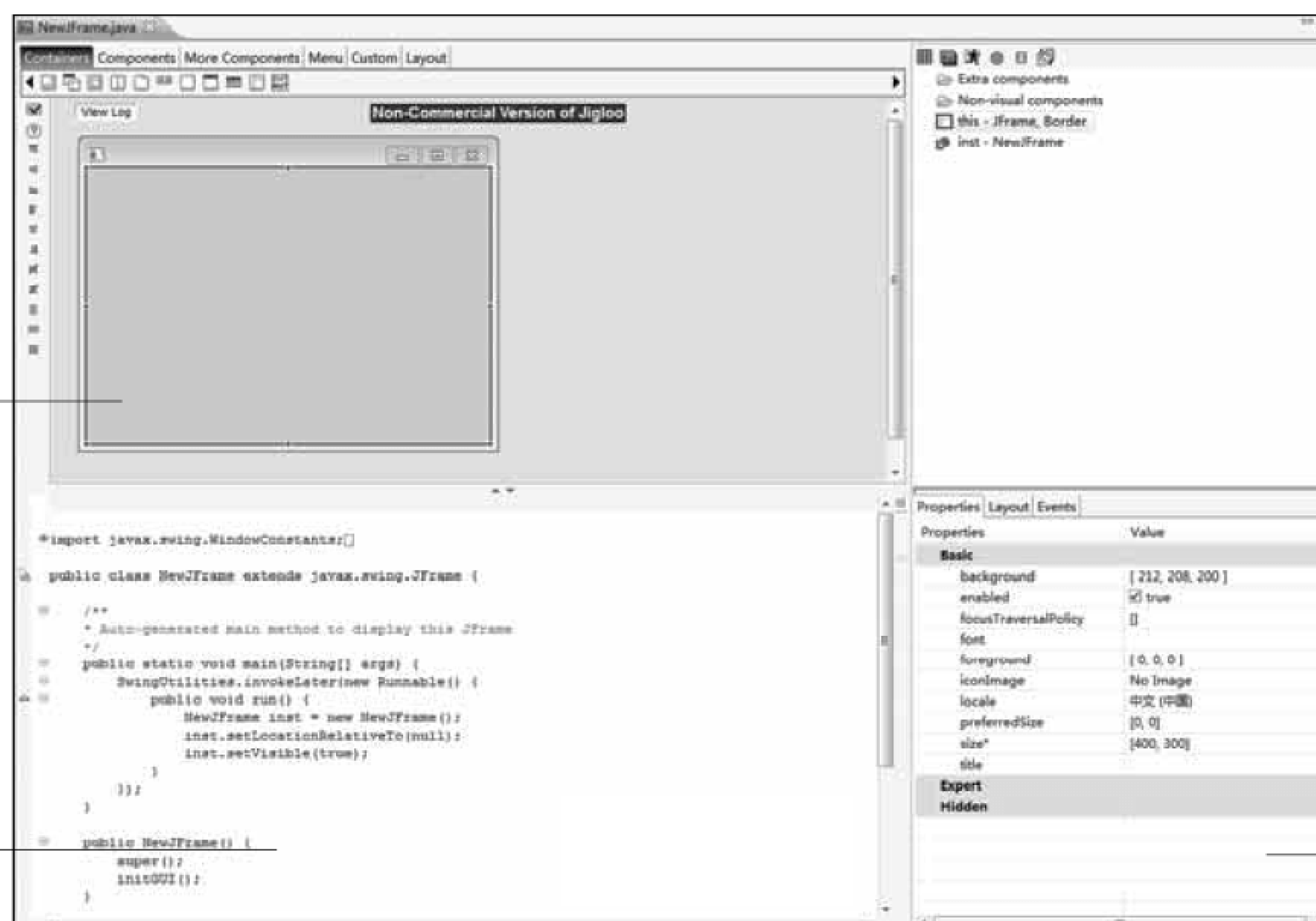
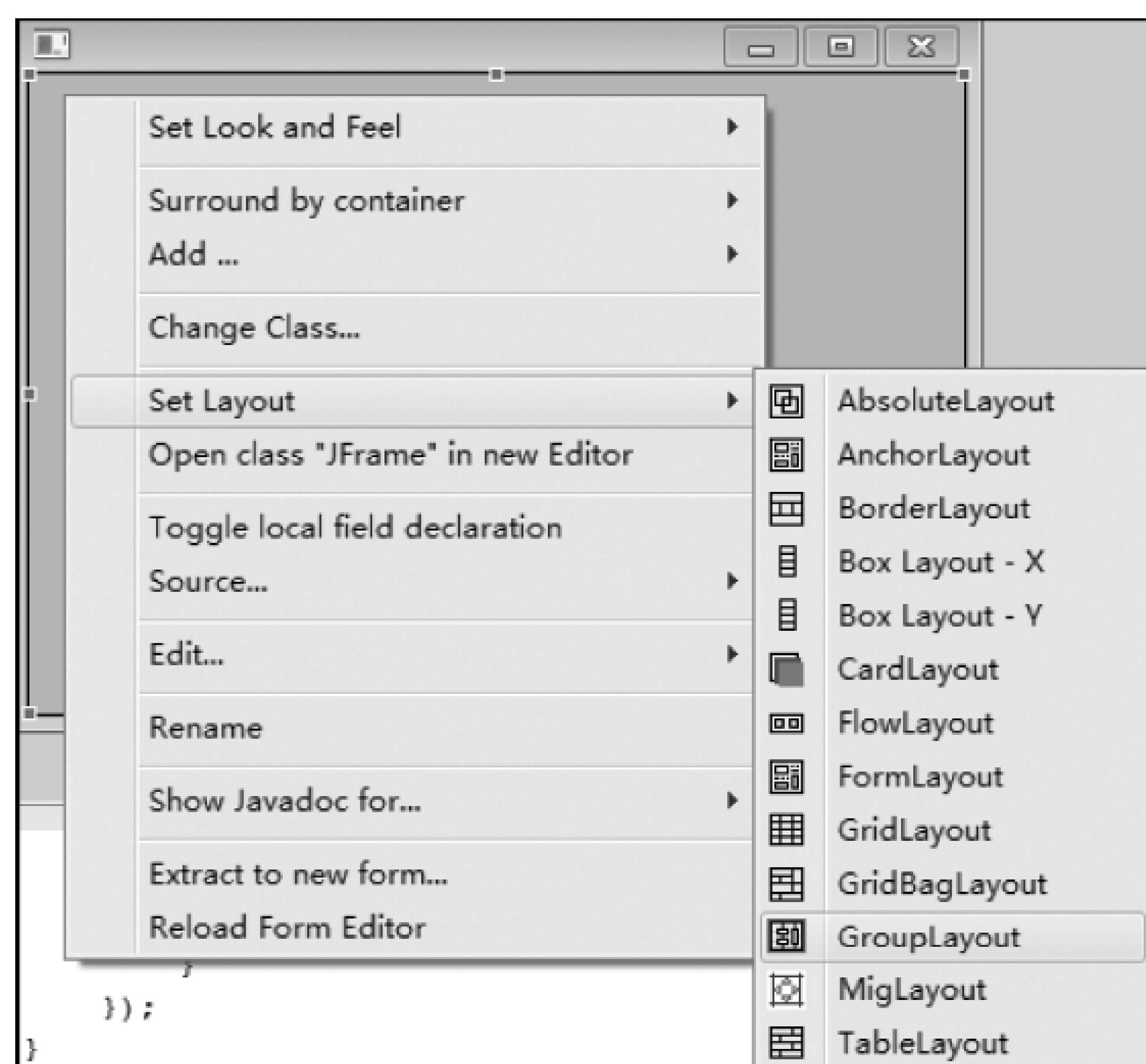
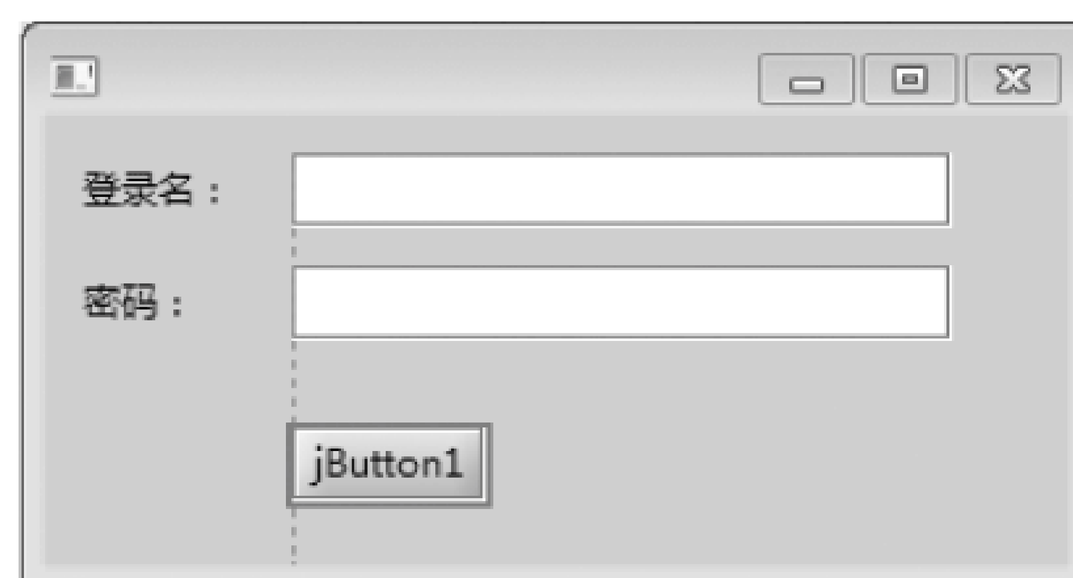


图 10-12　可视化布局界面



(a)



(b)

图 10-13　布局界面

域划分到不同的 Group(组)，再根据各个 Group(组)相对于水平轴(Horizontal)和垂直轴(Vertical)的排列方式来管理。之所以选择这种布局管理器，是因为这种布局管理器就是为了便于可视化工具进行布局而发明的。

接下来可以从工具栏上选择希望布局的组件，将其拖动到界面上，并调整大小设置属性，如图 10-13(b)所示。在界面布局的同时，Jigloo 会生成对应的 Java 代码。

使用 Jigloo 这样的工具就会发现，利用可视化工具进行界面布局是一件相当容易的事情。但本书为什么还要介绍 FlowLayout、BorderLayout 和 GridLayout 这些布局管理器呢？其原因在于本书希望读者知其然，也知其所以然。工具毕竟只是辅助，某些独特的界面效

果，仍然需要通过编写代码来实现。

10.3.6 实用案例

【例 10.7】 布局复杂界面。

复杂界面的布局往往非常复杂，单纯地使用一种布局管理器很难对 GUI 组件进行布局，因此在对复杂界面进行布局时往往需要组合使用多种布局管理器。通常使用 JPanel 类来辅助布局。JPanel 是一种不可见的容器，它可以通过 setLayout 方法设置布局方式，也可以用 add 方法添加 GUI 组件。JPanel 可以被布局在顶层容器上。如果 JPanel 上没有任何的 GUI 组件，则显示为空白区域。如果 JPanel 上有其他 GUI 组件，就会依据 JPanel 指定的布局方式进行布局。

下面是一个复杂界面的布局，顶层容器采用 GridLayout 方式布局(2 行、2 列)，包含了 4 个 JPanel 容器。4 个 JPanel 容器以从左到右、从上到下的顺序，分别采用 BorderLayout、FlowLayout、GridLayout 和无布局管理器布局进行布局。

```
ComplexLayoutDemo.java
import java.awt.BorderLayout;
import java.awt.FlowLayout;
import java.awt.GridLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
public class ComplexLayoutDemo extends JFrame {
    private JPanel panel1=new JPanel();
    private JPanel panel2=new JPanel();
    private JPanel panel3=new JPanel();
    private JPanel panel4=new JPanel();
    public ComplexLayoutDemo()
    {
        setSize(500, 500);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //对 panel1 进行布局
        layoutPanel1();
        //对 panel2 进行布局
        layoutPanel2();
        //对 panel3 进行布局
        layoutPanel3();
        //对 panel4 进行布局
        layoutPanel4();
        //对顶层容器进行布局,采用 GridLayout,2 行、2 列
        setLayout(new GridLayout(2,2));
        add(panel1);
        add(panel2);
```

```
        add(panel3);
        add(panel4);
    }
    private void layoutPanel1() {
        JButton north= new JButton("北");
        JButton south= new JButton("南");
        JButton east= new JButton("东");
        JButton west= new JButton("西");
        JButton center= new JButton("中");
        //panel1 采用 BorderLayout 布局
        panel1.setLayout(new BorderLayout());
        panel1.add(north,BorderLayout.NORTH);
        panel1.add(south,BorderLayout.SOUTH);
        panel1.add(east,BorderLayout.EAST);
        panel1.add(west,BorderLayout.WEST);
        panel1.add(center,BorderLayout.CENTER);
    }
    private void layoutPanel2() {
        JButton button1= new JButton("按键 1");
        JButton button2= new JButton("按键 2");
        JButton button3= new JButton("按键 3");
        JButton button4= new JButton("按键 4");
        //panel2 采用 FlowLayout 布局
        panel2.setLayout(new FlowLayout());
        panel2.add(button1);
        panel2.add(button2);
        panel2.add(button3);
        panel2.add(button4);
    }
    private void layoutPanel3() {
        JButton button1= new JButton("按键 1");
        JButton button2= new JButton("按键 2");
        JButton button3= new JButton("按键 3");
        JButton button4= new JButton("按键 4");
        //panel3 采用 GridLayout 布局,2 行、2 列
        panel3.setLayout(new GridLayout(2,2));
        panel3.add(button1);
        panel3.add(button2);
        panel3.add(button3);
        panel3.add(button4);
    }
    private void layoutPanel4() {
        JButton button= new JButton("按键");;
        JTextField textField= new JTextField("文本框");
        //panel2 采用无布局管理器布局
```

```
            panel4.setLayout(null);
            button.setLocation(20, 20);
            button.setSize(100, 20);
            textField.setBounds(20,50,200,100);
            panel4.add(button);
            panel4.add(textField);
        }
        public static void main(String[] args) {
            ComplexLayoutDemo frame=new ComplexLayoutDemo();
            frame.setVisible(true);
        }
    }
```

程序运行结果如图 10-14 所示。

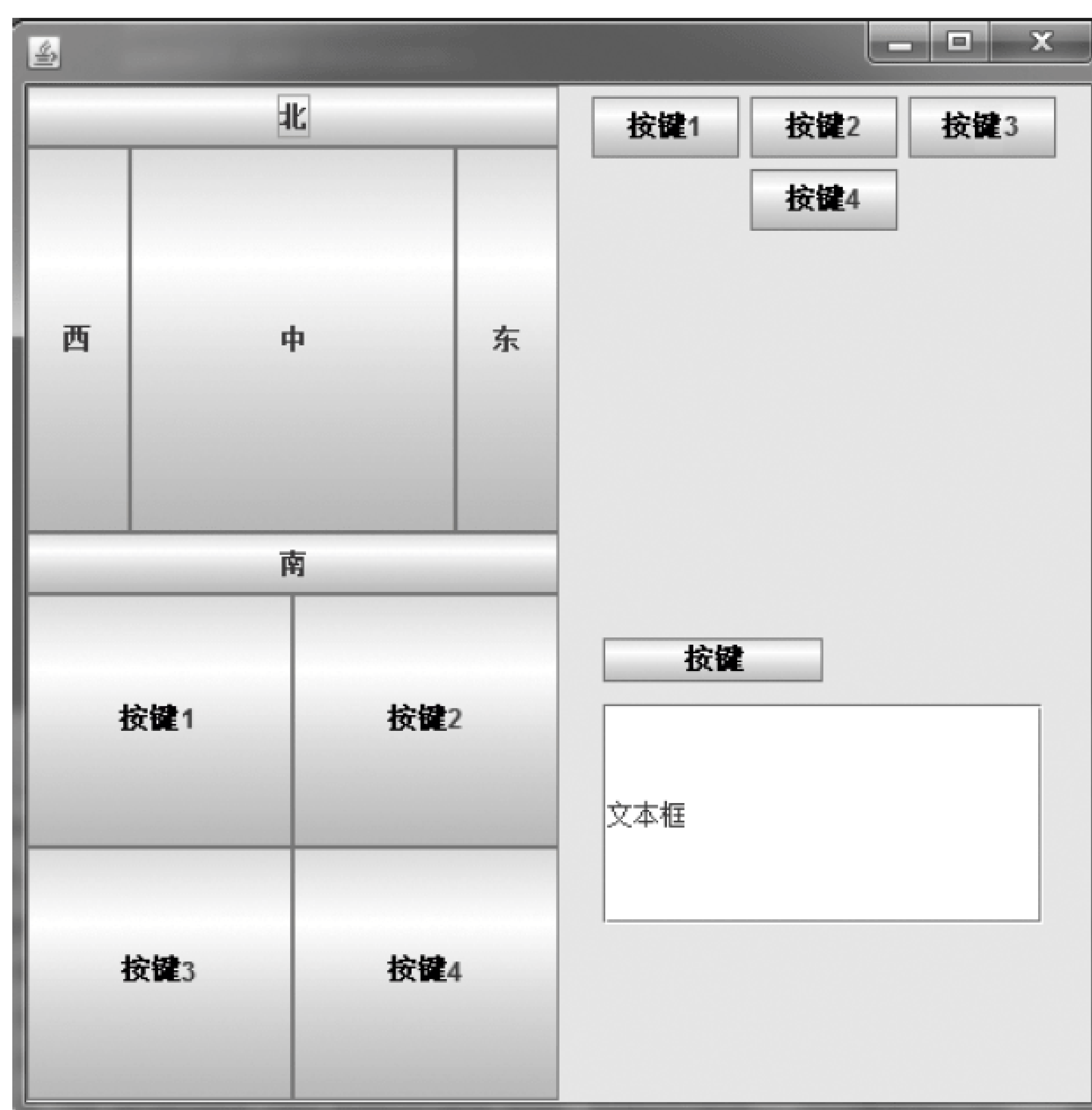


图 10-14　复杂界面布局

10.4　响应用户事件

10.4.1　事件处理的基本过程

响应用户事件是 GUI 程序设计的重要内容。用户事件包括鼠标的单击、GUI 组件值的改变、焦点的获取或者丢失、键盘输入等。不同的 GUI 组件可以响应不同类型的事件。当用户事件发生时，该事件会涉及以下 3 个方面的对象。

(1) 事件源(Source)：产生事件的对象。这类对象包括 GUI 组件，例如 JButton、JTextField 等。

(2) 事件(Event)：发生在用户界面上的，由用户交互行为所产生的一种效果。这类对

象由 java. awt. AWTEvent 及其子类构成,例如 KeyEvent、MouseEvent 等。

(3) 事件监听器(Listener):接受事件并对其进行处理的对象。这类对象由实现了 java. util. EventListener 接口的类构成,例如 ActiontListener、MouseListener 等。

例如,当用户用鼠标单击一个按键时,鼠标单击动作会触发事件,按键就是事件源,而对该事件进行处理的对象就是事件监听器。通常一个事件源可以触发多种类型的事件,每一个事件可以由一个或者多个事件监听器进行处理。Swing 中每个 GUI 组件(JComponent 及其子类)都有若干名为 addXXXlistener 的方法,例如 JButton 类有 addActionListener、addKeyListener 等 addXXXlistener 方法。这类方法被用于注册特定事件的监听器。其中 addActionListener 用于注册按键单击的事件监听器,addKeyListener 用于注册键盘输入的事件监听器。下面以例 10.1 中 JButton 组件的单击事件为例,说明如何编写事件处理程序。例 10.1 中涉及事件处理的代码如下:

```
button.addActionListener(new ActionListener(){
        public void actionPerformed(ActionEvent event) {
            JOptionPane.showMessageDialog(null, "单击了按键!");
        }

    });
```

该事件的处理过程,首先调用 JButton 的 addActionListener 方法,添加一个事件监听器。负责处理按键单击事件的监听器是实现了 ActionListener 接口的子类。例 10.1 利用匿名类实现了该接口。ActionListener 接口包含一个唯一的方法,即:

```
public void actionPerformed(ActionEvent e)
```

程序通过实现该方法来处理按键单击事件。actionPerformed 方法的参数 e 的类型为 ActionEvent。ActionEvent 是对应单击事件的类。通过调用对象 e 的方法可以获取事件的相关属性。例如,调用 getSource 方法将返回事件发生的对象。actionPerformed 方法的实现可以简单,也可以复杂,具体的情况要根据程序所需要实现的事件处理来确定。例 10.8 给出了另一个按键事件处理程序。

【例 10.8】 一个简单的按键处理程序。

```
import java.awt.FlowLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.JButton;
import javax.swing.JFrame;
public class EventDemo extends JFrame{
    JButton button= new JButton("点我!");
    public EventDemo() {
        setSize(300,300);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        //设置按键事件,使用了匿名类
        button.addActionListener(new ActionListener(){
```

```
                public void actionPerformed(ActionEvent e) {
                    //获取被单击的按键
                    JButton clickedButton= (JButton) e.getSource();
                    //改变被单击按键的标题
                    clickedButton.setText("我被点了!");
                }
            });
            setLayout(new FlowLayout());
            add(button);
        }
        public static void main(String[] args) {
            EventDemo frame= new EventDemo();
            frame.setVisible(true);
        }
    }
```

在例 10.8 中匿名类实现了 ActionListener 接口,并在 actionPerformed 中通过 ActionEvent 的 getSource 方法获取被单击的按键,然后调用按键的 setText 方法替换原有按键的标题。图 10-15(a)是单击按键前的界面,图 10-15(b)是单击按键之后的界面。事件监听器不仅可以实现为匿名类,还可以用内部类、外部类,也可以用主类。这几种方式实现的监听器在对事件的处理上并没有本质的区别。

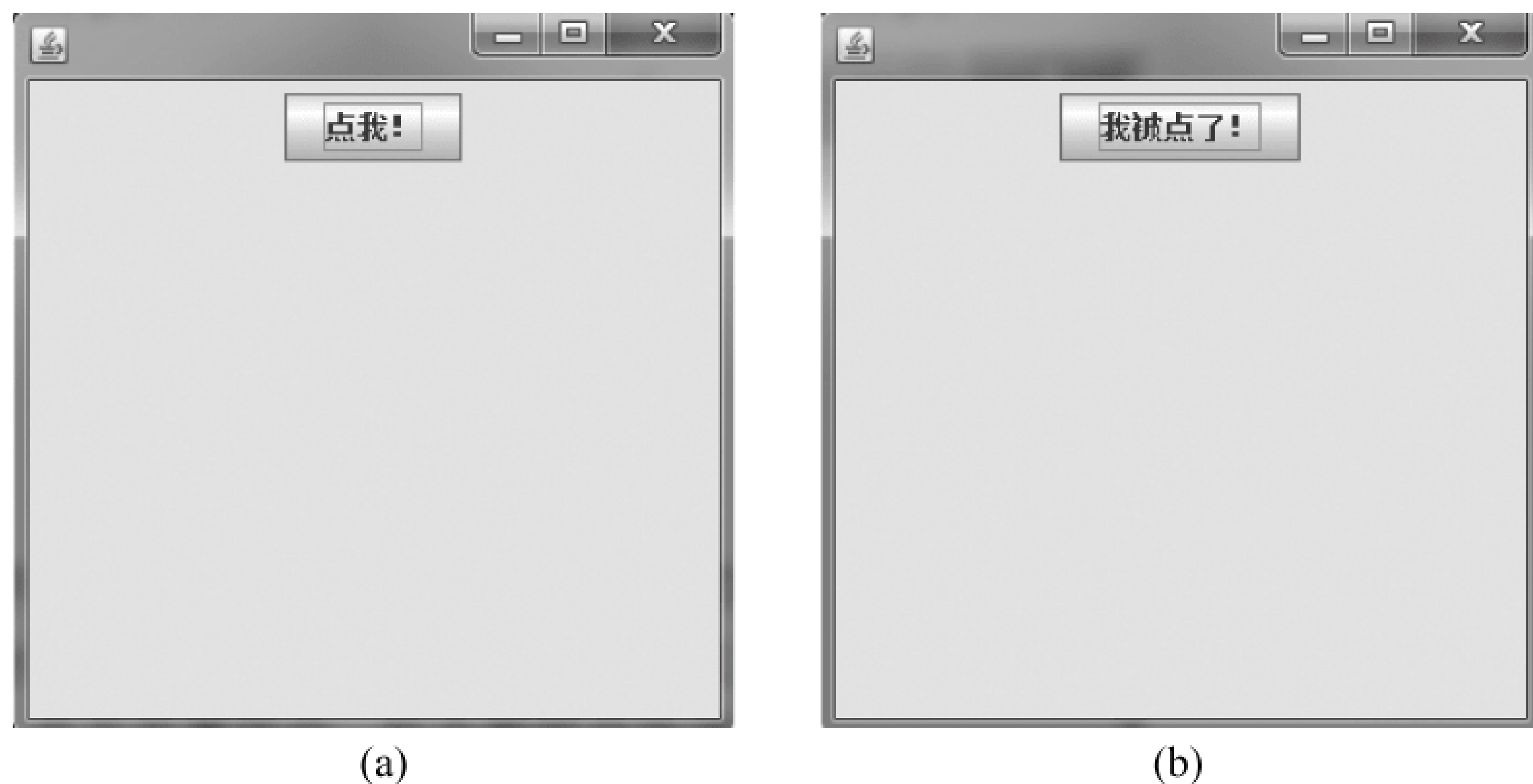


图 10-15　按键处理程序

10.4.2　常用事件与事件监听器类

Swing 事件类的基类是 java.awt.AWTEvent,它是 java.util.EventObject 的子类。基类 EventObject 定义了方法 getSource,该方法返回产生或触发事件的对象。AWTEvent 定义了方法 getID,该方法的返回值用来区别用同一个事件类所代表的不同类型的事件。AWTEvent 下常见的事件类及其结构关系,如图 10-16 所示。不同的事件类,除了拥有父类的方法,还定义一些与该事件相关的方法。例如,MouseEvent 有方法 getX、getY 和 getClickCount,用于返回鼠标事件产生的坐标位置和鼠标单击的次数。

每个事件类都有对应的事件监听器接口,接口中定义了事件发生时可调用的方法。在

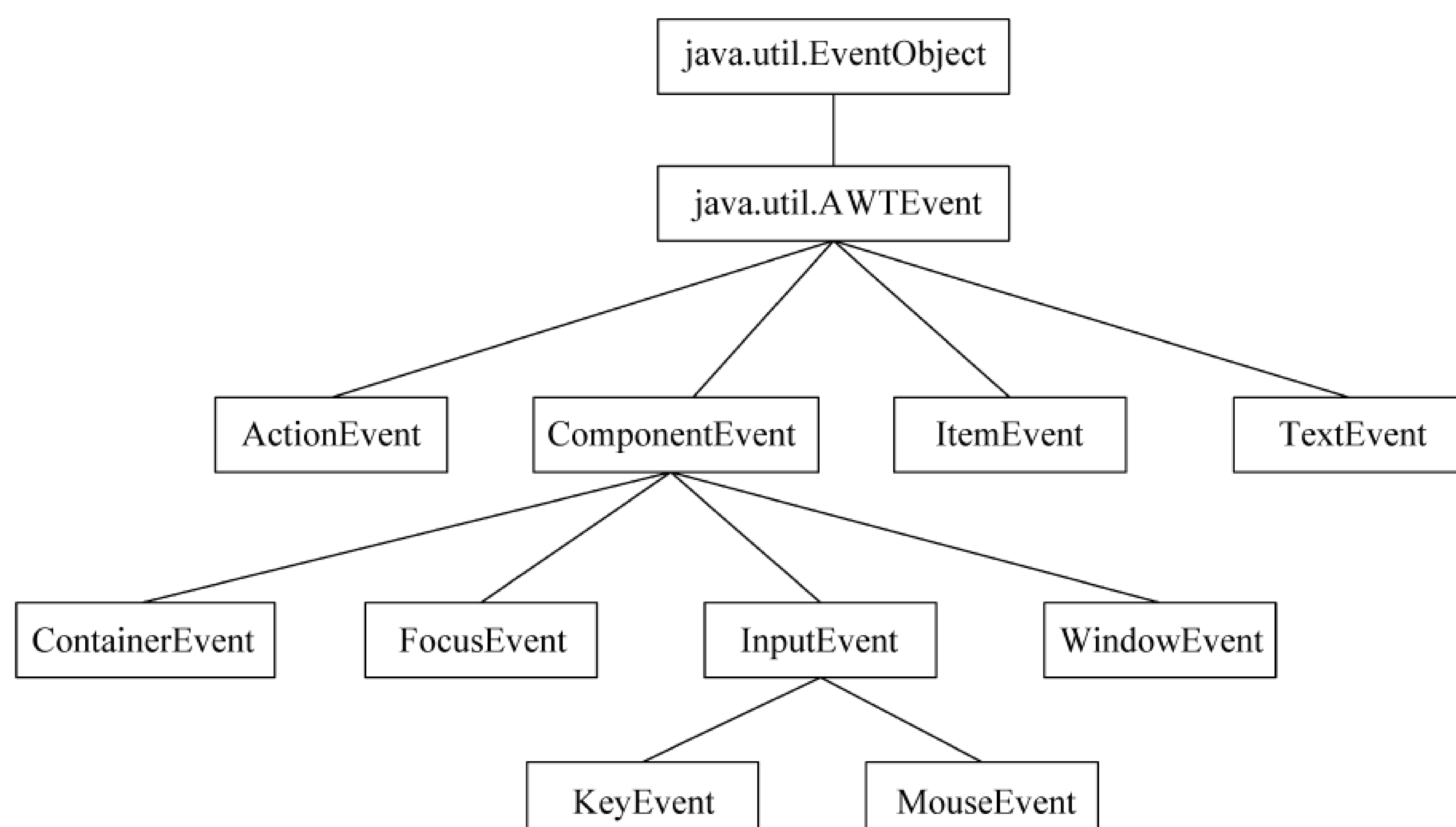


图 10-16　常见的事件类及其结构关系

处理用户事件时，需要根据事件类型向 GUI 组件注册对应的事件监听器(调用对应的 addXXXListener 方法)，也就是要实现相应的事件监听接口。表 10-2 给出了图 10-16 所示事件类对应的事件监听器接口，以及该类事件产生的原因。

表 10-2　常见事件监听器

事件类/监听器接口	接口中声明的方法	事件产生原因
ComponentEvent ComponentListener	componentMoved(ComponentEvent e)	移动组件时
	componentHidden(ComponentEvent e)	隐藏组件时
	componentResized(ComponentEvent e)	改变组件大小时
	componentShown(ComponentEvent e)	显示组件时
ContainerEvent ContainerListener	componentAddedComponentEvent e)	添加组件时
	componentRemovedComponentEvent e)	移动组件时
WindowEvent WindowListener	WindowOpened(WindowEvent e)	打开窗口时
	WindowActivated(WindowEvent e)	激活窗口时
	WindowDactivated(WindowEvent e)	窗口失去焦点时
	WindowClosing(WindowEvent e)	关闭窗口时
	WindowClosed(WindowEvent e)	关闭窗口后
	WindowIconified(WindowEvent e)	窗口最小化时
	WindowDeiconified(WindowEvent e)	当窗口从最小恢复到正常大小时
ActionEvent ActionListener	ActionPerformed(ActionEvent e)	单击按钮，文本框中单击回车键，双击列表框选择菜单项等

续表

事件类/监听器接口	接口中声明的方法	事件产生原因
TextEvent TextListener	textValueChanged(TextEvent e)	文本框中修改内容
ItemEvent ItemListener	ItemStateChanged(ItemEvent e)	选择复选框、选择框，单击列表框，选中带复选框的菜单项
MouseEvent MouseMotionListener	mouseDragged(MouseEvent e)	鼠标拖动时
	mouseMoved(MouseEvent e)	鼠标移动时
MouseEvent MouseListener	mouseClicked(MouseEvent e)	单击鼠标时
	mouseEntered(MouseEvent e)	鼠标进入时
	mouseExited(MouseEvent e)	鼠标离开时
	mousePressed(MouseEvent e)	按下鼠标时
	mouseReleased(MouseEvent e)	放开鼠标时
KeyEvent KeyListener	keyPresssed(KeyEvent e)	按下键盘时
	keyReleased(KeyEvent e)	释放键盘时
	keyTyped(KeyEvent e)	按了键盘上的按钮时
FocusEvent FocusListener	focusGained(FocusEvent e)	获得焦点时
	focusLost(FocusEvent e)	失去焦点时

10.4.3 键盘与鼠标事件

键盘事件和鼠标事件是 GUI 程序中最常见的两类事件。如表 10-2 所示，与键盘事件相关的监听器是 KeyListener，与鼠标事件相关的监听器包括 MouseListener、MouseMotionListener 和 MouseWheelListener。为了处理相应的事件，需要调用相应的 addXXXlistener 方法，添加相应的事件监听器。

在处理事件时，事件监听器接口可以采用几个方法，例如 KeyListener 有 3 个方法，如果只关心“按下键盘”按钮，而不关心其他两种情况，也必须实现 KeyListener 的 3 个方法。为此 Java 提供了事件适配器类来简化这个问题，java.awt.event 包中定义的事件适配器类包括以下几个：

(1) ComponentAdapter 类：实现了 ComponentListener。

(2) ContainerAdapter 类：实现了 ContainerListener。

(3) FocusAdapter 类：实现了 FocusListener。

(4) KeyAdapter 类：实现了 KeyListener。

(5) MouseAdapter 类：实现了 MouseListener 和 MouseMotionListener。

(6) WindowAdapter 类：实现了 WindowListener。

使用适配器的过程与实际监听器类似，但只须重写事件处理所关心的方法，其他方法不用实现，这样就简化了程序代码。同时，适配器是一个类，而不是接口，但使用适配器时仍然

必须继承对应的适配器类。键盘和鼠标事件所对应的适配器类分别为 KeyAdapter 和 MouseAdapter。例 10.9 和例 10.10 简要演示了键盘和鼠标事件的处理。出于简化程序的目的,示例使用了 KeyAdapter 和 MouseAdapter。

【例 10.9】 键盘事件处理程序。

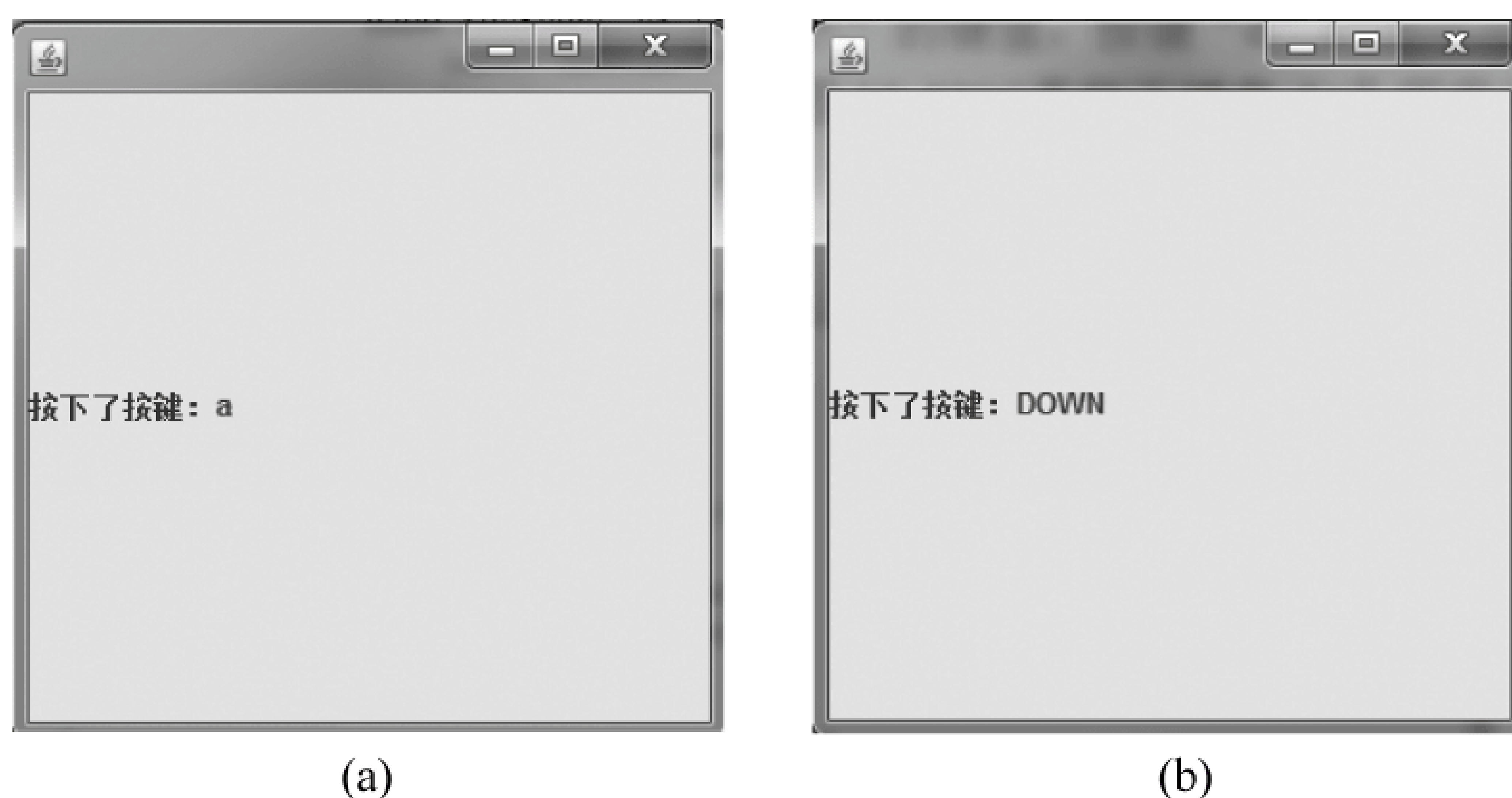
```
import java.awt.BorderLayout;
import java.awt.event.KeyAdapter;
import java.awt.event.KeyEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class KeyEventDemo extends JFrame{
    JLabel label=new JLabel("按下了按键:");
    public KeyEventDemo() {
        setSize(300,300);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.addKeyListener(new KeyAdapter(){
            public void keyPressed(KeyEvent event) {
                switch(event.getKeyCode())
                {
                    case KeyEvent.VK_UP:
                        label.setText("按下了按键:UP");
                        break;
                    case KeyEvent.VK_DOWN:
                        label.setText("按下了按键:DOWN");
                        break;
                    case KeyEvent.VK_LEFT:
                        label.setText("按下了按键:LEFT");
                        break;
                    case KeyEvent.VK_RIGHT:
                        label.setText("按下了按键:RIGHT");
                        break;
                    default:
                        label.setText("按下了按键:"+event.getKeyChar());
                }
            }
        });
        setLayout(new BorderLayout());
        add(label,BorderLayout.CENTER);
    }
    public static void main(String[] args) {
        KeyEventDemo frame=new KeyEventDemo();
        frame.setVisible(true);
    }
}
```

例 10.9 只重写了 KeyAdapter 的 keyPresssed 方法，用于处理按下键盘的事件。在事件处理过程中使用了 KeyEvent。KeyEvent 是对键盘事件的封装，在 KeyEvent 中有以下几个常用的方法。

（1）getKeyChar()：获取触发事件按键对应的字符。例如，当按下按键 d 时，所获得的字符就是"d"。

（2）getKeyCode()：获取触发事件按键对应的键值。所谓键值，在 KeyEvent 中有若干常量与之对应。例如，KeyEvent.VK_UP 对应方向键“上”的键值，KeyEvent.VK_DOWN 对应方向键“下”的键值，按键 d 的键值是 KeyEvent.VK_D。

运行例 10.10 程序将显示如图 10-17 所示窗体。图 10-17(a)是按下键盘 a 所显示的结果，图 10-17(b)是按下方向键“下”所显示的结果。



(a) (b)

图 10-17 键盘响应示例

【例 10.10】 鼠标事件处理程序。

```
import java.awt.BorderLayout;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import javax.swing.JFrame;
import javax.swing.JLabel;
public class MouseEventDemo extends JFrame{
    JLabel label=new JLabel("");
    public MouseEventDemo() {
        setSize(300,300);
        setLocation(400, 400);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.addMouseListener(new MouseAdapter(){
            public void mouseClicked(MouseEvent event) {
                label.setText("鼠标在"+event.getX()+","+event.getY()+"进行了
                单击");
            }
        });
        this.addMouseMotionListener(new MouseAdapter(){
```

```
                public void mouseMoved(MouseEvent event) {
                    label.setText("鼠标移动到了"+event.getX()+","+event.getY());
                }
            });
            setLayout(new BorderLayout());
            add(label,BorderLayout.CENTER);
        }
        public static void main(String[] args) {
            MouseEventDemo frame=new MouseEventDemo();
            frame.setVisible(true);
        }
    }
```

例 10.10 使用 MouseAdapter 为窗体添加了 MouseListener 和 MouseMotionListener 两种类型的监听器。在事件处理中用 MouseEvent 的 getX 和 getY 方法获得事件发生时鼠标所在的位置。图 10-18(a)是鼠标单击时的事件响应,图 10-18(b)是鼠标移动时的事件响应。

例 10.10 程序运行结果如图 10-18 所示。

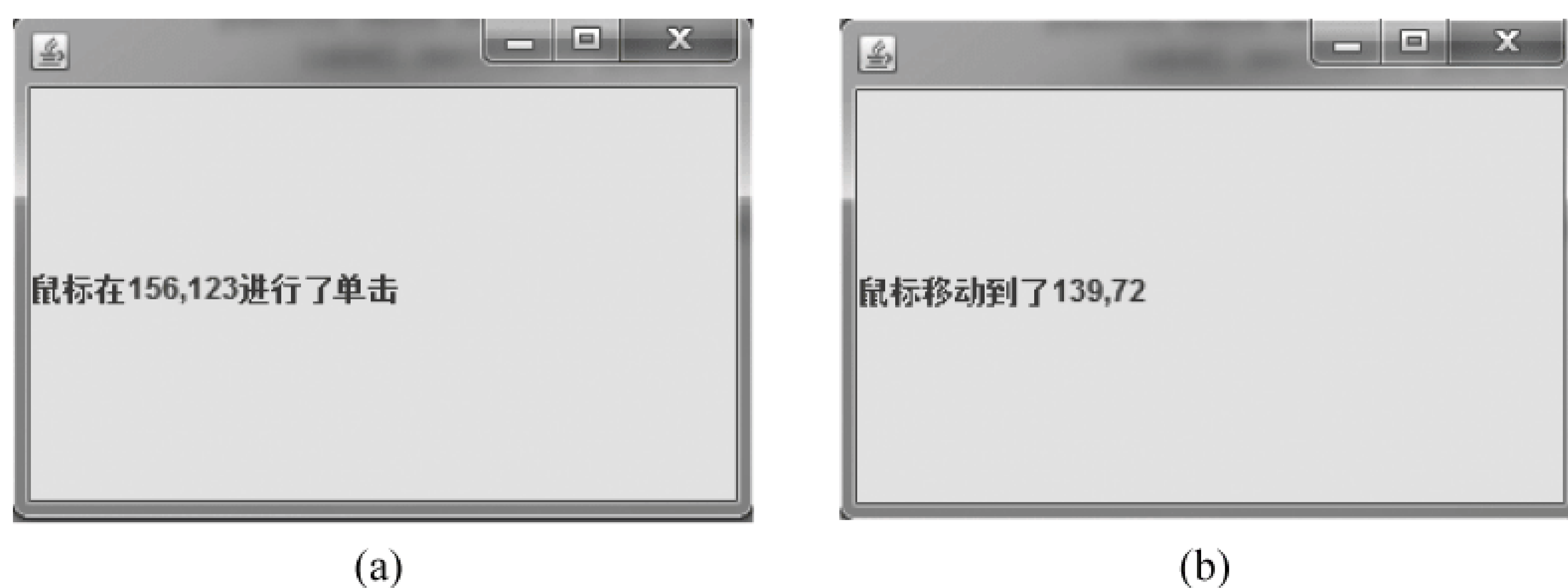


图 10-18　鼠标处理事件

10.4.4　实用案例

【例 10.11】　用鼠标绘图。

使用鼠标事件和窗体事件实现用鼠标绘图。当鼠标在窗体上移动时,同时绘制出图像。本例使用了 BufferedImage 和 Graphics 类,这两个类主要用于图形绘制。鉴于本书篇幅所限,这里不对它们进行详细描述,如果想深入了解 Java 的图形绘制,可以参考其他书籍。

```
import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ComponentAdapter;
import java.awt.event.ComponentEvent;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;

import javax.swing.JFrame;
```

```
public class PaintFrame extends JFrame {
    //记录绘图内容
    private BufferedImage canvas;
    //记录上一次鼠标的 x 位置
    private int lastX=-1;
    //记录上一次鼠标的 y 位置
    private int lastY=-1;

    public PaintFrame() {
        //窗口大小改变时间
        this.addComponentListener(new ComponentAdapter() {
            public void componentResized(ComponentEvent e) {
                BufferedImage newCanvas=new BufferedImage(getWidth(),
                getHeight(), BufferedImage.TYPE_INT_RGB);
                if (canvas !=null) {
                    Graphics g=newCanvas.getGraphics();
                    g.drawImage(canvas, 0, 0, null);
                }
                canvas=newCanvas;
            }
        });
        //鼠标移动事件
        this.addMouseMotionListener(new MouseAdapter() {
            public void mouseMoved(MouseEvent e) {
                int x=e.getX();
                int y=e.getY();
                if (lastX==-1 && lastY==-1) {
                    lastX=x;
                    lastY=y;

                } else {
                    Graphics g=canvas.getGraphics();
                    g.setColor(Color.RED);
                    g.drawLine(lastX, lastY, x, y);
                    lastX=x;
                    lastY=y;
                    //重绘触发 paint
                    repaint();

                }
            }

        });

    }
    //在窗口上绘制图片
    public void paint(Graphics g) {
```

```
            g.drawImage(canvas, 0, 0, null);
        }

        public static void main(String[] args) {
            PaintFrame frame=new PaintFrame();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setSize(500, 500);
            frame.setVisible(true);
        }
    }
```

例 10.11 程序运行结果如图 10-19 所示。

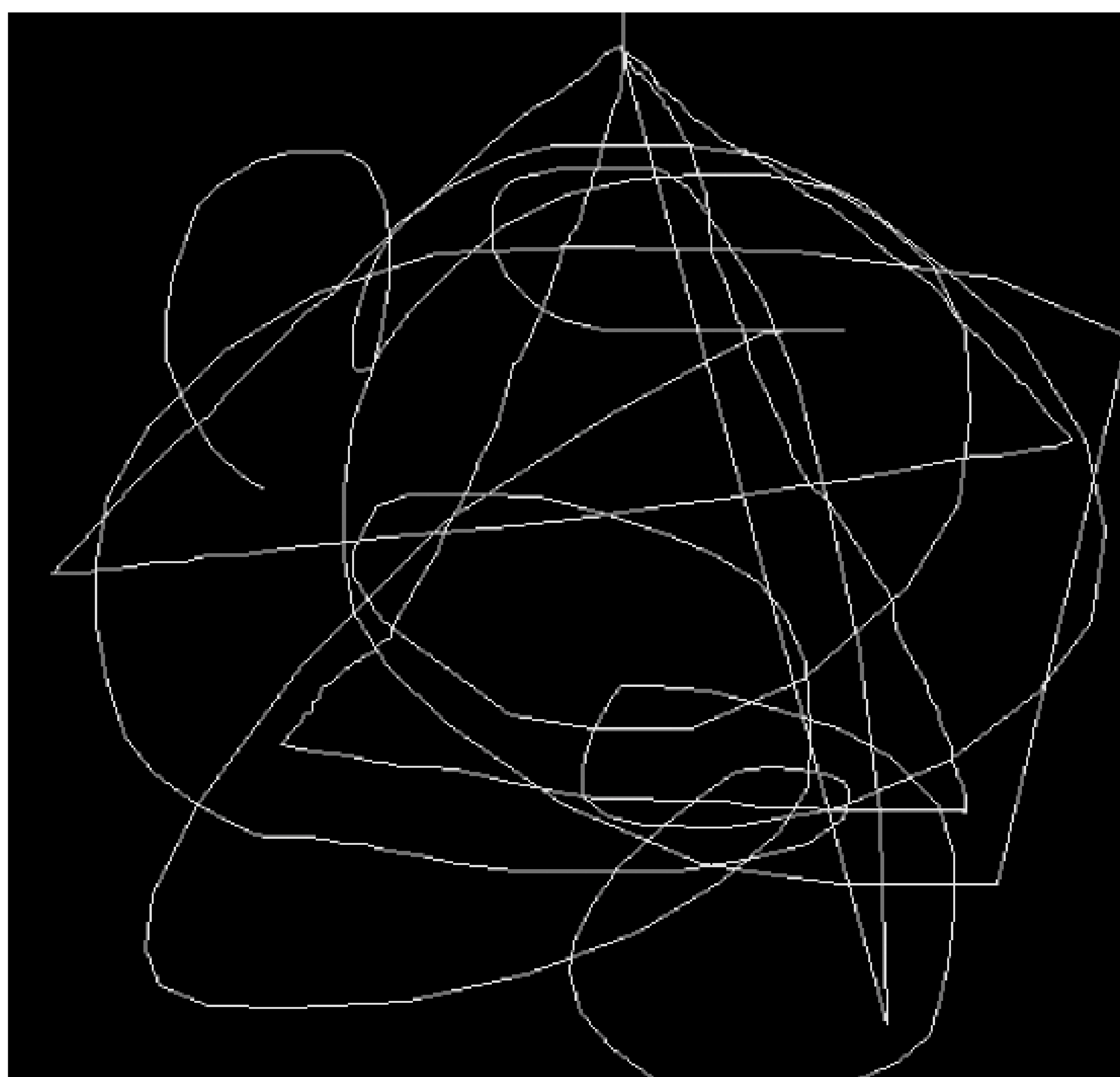


图 10-19　例 10.11 程序运行结果

10.5　高级组件 JTree 和 JTable

10.5.1　JTree 组件

JTree 是 Swing 中的高级界面组件,它用于将数据显示为树形结构。其常用的构造方法如下。

(1) JTree():返回一棵默认的示例 JTree。

(2) JTree(Hashtable<?,?>value):从 Hashtable 创建 JTree,此方法构造的 JTree 不显示根结点。

(3) JTree(Object[] value):用 Object 数组的每个元素作为 JTree 的子结点,此方法构造的 JTree 不显示根结点。

(4) JTree(TreeModel newModel):使用指定的 TreeModel 数据模型创建 JTree,此方法构造的 JTree 显示根结点。

(5) JTree(TreeNode root):指定 root 作为 JTree 的根结点,并用 root 的子结点作为

JTree 的子结点。

(6) JTree(Vector<?>value)：用 Vector 的元素作为 JTree 的子结点，此方法构造的 JTree 不显示根结点。

下面来看一个简单的示例。

【例 10.12】 建立一棵默认的树。

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
public class SimpleTree extends JFrame{
    public SimpleTree() {
        JTree tree=new JTree();
        JScrollPane scrollPane=new JScrollPane();
        scrollPane.setViewportView(tree);
        add(scrollPane);
    }
    public static void main(String[] args) {
        SimpleTree frame=new SimpleTree();
        frame.setTitle("SimpleTree");
        frame.setSize(300, 300);
        frame.setVisible(true);
    }
}
```

例 10.12 创建了一棵默认的树，如图 10-20 所示。但通常希望在使用树时，树结点的名称、结构都是自定义的。系统默认的树显然不能够满足需求。例 10.13 通过 HashTable 构造 JTree。

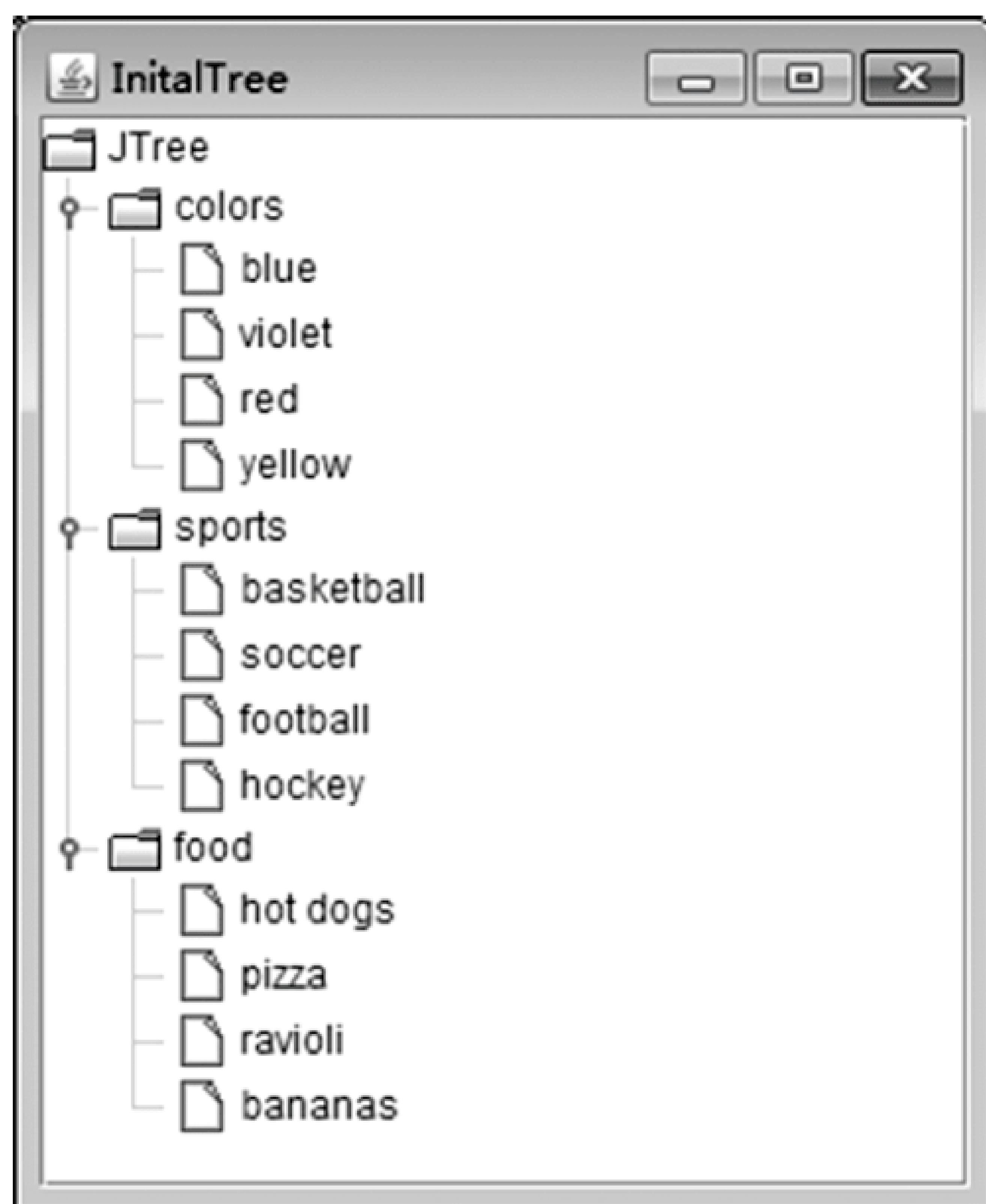


图 10-20 建立一棵默认的树

【例 10.13】 使用 HashTable 构造树。

```
import java.util.Hashtable;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
public class HashConstructTree extends JFrame{
  public HashConstructTree(){
    String[] s1={"A","B","C"};
    String[] s2={"a","b","c"};
    Hashtable<String,String[]> hashtable=new Hashtable<String,String[]>();
    hashtable.put("大写字母",s1);
    hashtable.put("小写字母",s2);
    JTree tree=new JTree(hashtable);
    JScrollPane scrollPane=new JScrollPane();
    scrollPane.setViewportView(tree);
      add(scrollPane);
  }
  public static void main(String[] args){
    HashConstructTree frame=new HashConstructTree();
    frame.setTitle("HashConstructTree");
    frame.setSize(300, 300);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }
}
```

例 10.13 创建的树如图 10-21 所示,在实际使用中 HashTable 可嵌套使用以构造复杂结构的树。

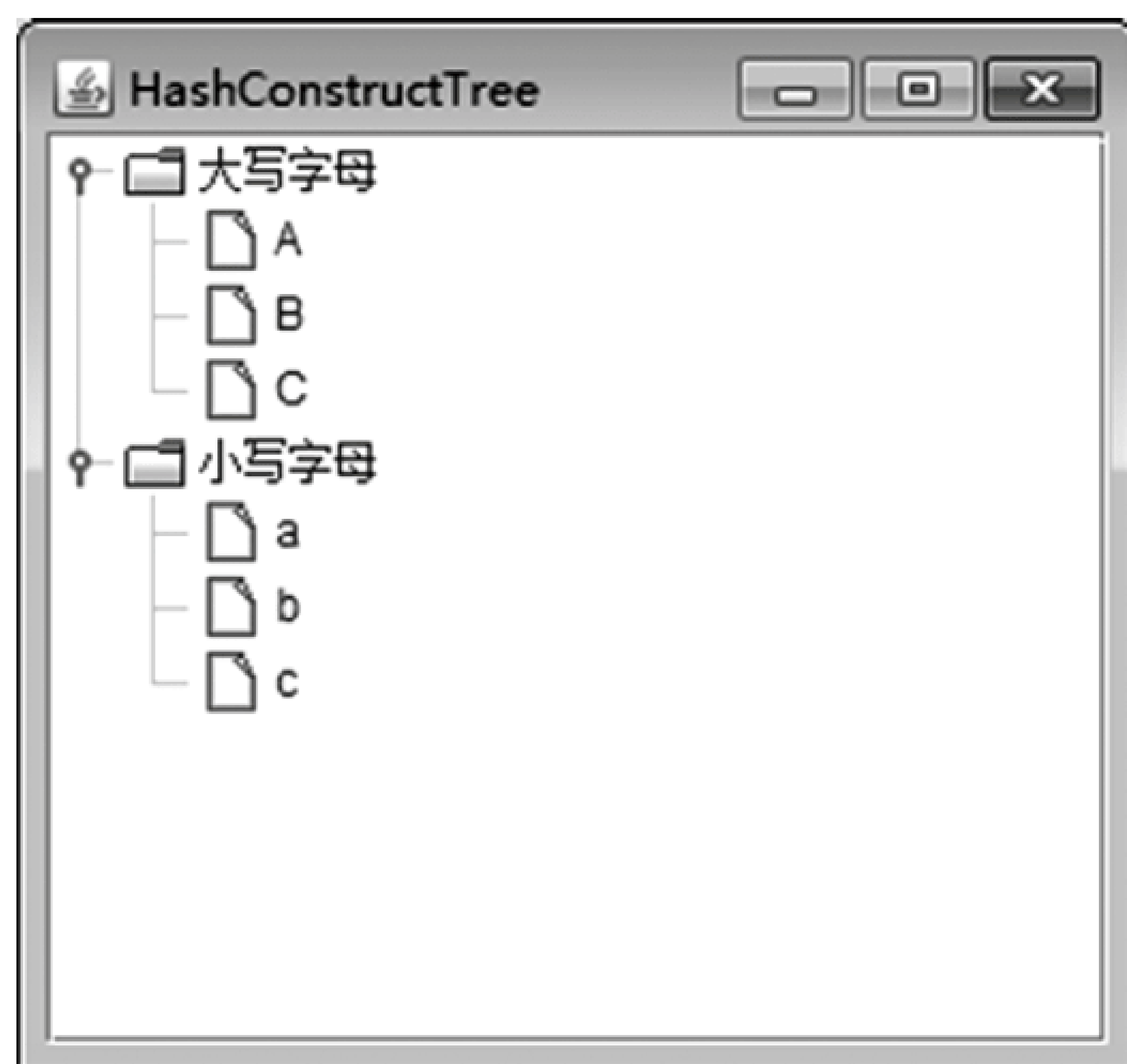


图 10-21 使用 HashTable 建立的树

使用 TreeNode 构造树，是所有方法中最为常用的一种。在这种方法中，JTree 上的每一个结点就是一个 TreeNode 对象。TreeNode 是一个接口，里面定义了若干有关结点的方法，如判断是否为树叶结点、有几个子结点、获得父结点等。在实际的应用上，一般不会直接实现 TreeNode，而是采用 DefaultMutableTreeMode 类。DefaultMutableTreeMode 类实现了 MutableTreeNode 接口，并提供了其他许多实用的方法。而 MutableTreeNode 本身也是一个接口，并且继承了 TreeNode。MutableTreeNode 接口主要是定义一些结点的处理方式，例如新增结点、删除结点、设置结点等。例 10.14 演示了 DefaultMutableTreeNode 的用法，其运行效果如图 10-22 所示。

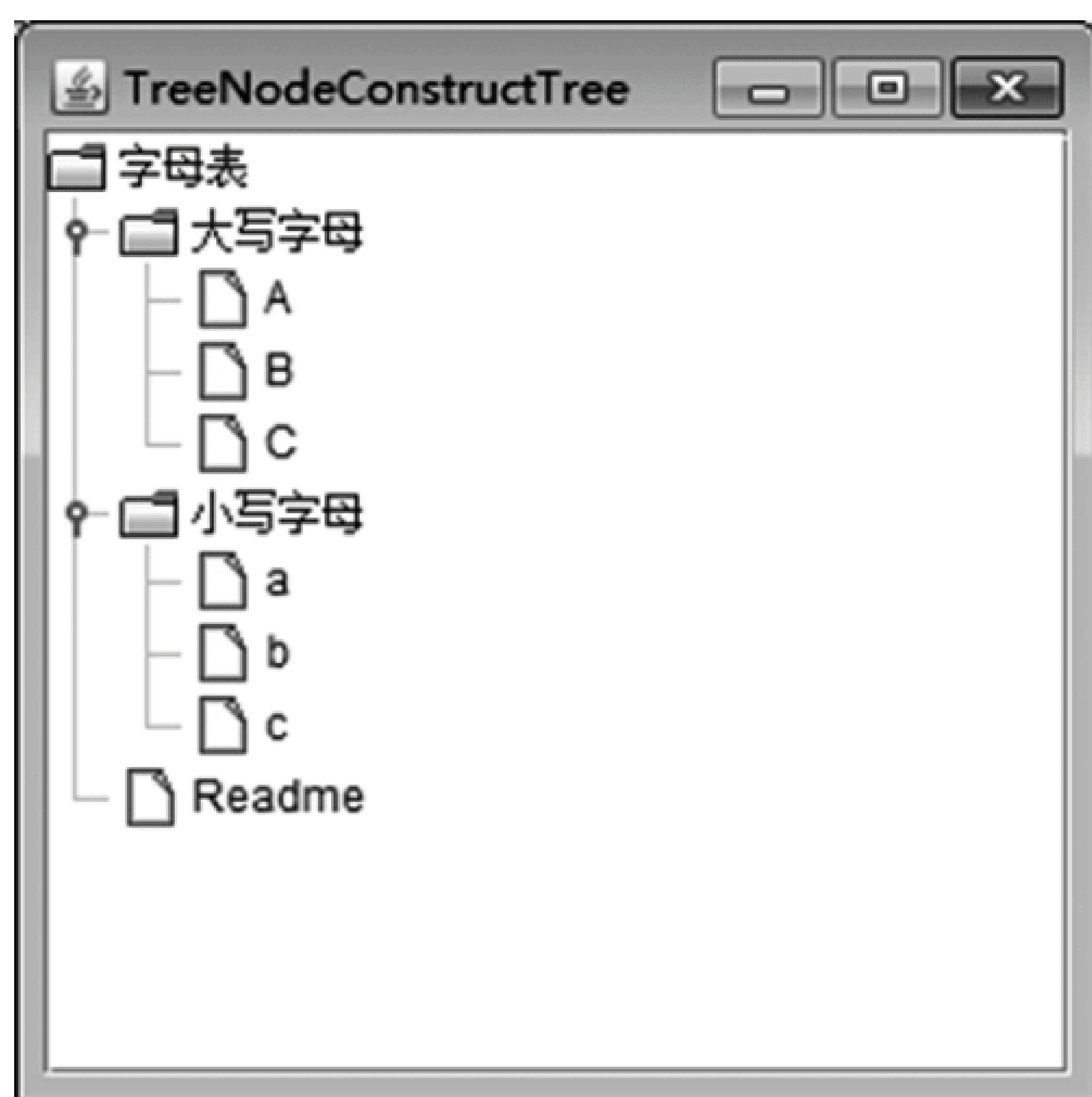


图 10-22　使用 DefaultMutableTreeNode 构造 JTree

【例 10.14】 使用 DefaultMutableTreeNode 构造 JTree。

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTree;
import javax.swing.tree.DefaultMutableTreeNode;
public class TreeNodeConstructTree extends JFrame{
    public TreeNodeConstructTree() {
        DefaultMutableTreeNode root=new DefaultMutableTreeNode("字母表");
        DefaultMutableTreeNode node1=new DefaultMutableTreeNode("大写字母");
        DefaultMutableTreeNode node2=new DefaultMutableTreeNode("小写字母");
        DefaultMutableTreeNode node3=new DefaultMutableTreeNode("Readme");
        root.add(node1);
        root.add(node2);
        root.add(node3);
        node1.add(new DefaultMutableTreeNode("A"));
        node1.add(new DefaultMutableTreeNode("B"));
        node1.add(new DefaultMutableTreeNode("C"));
        node2.add(new DefaultMutableTreeNode("a"));
        node2.add(new DefaultMutableTreeNode("b"));
```

```
            node2.add(new DefaultMutableTreeNode("c"));
            JTree tree=new JTree(root);
            JScrollPane scrollPane=new JScrollPane();
            scrollPane.setViewportView(tree);
            add(scrollPane);
        }
        public static void main(String[] args) {
            TreeNodeConstructTree frame=new TreeNodeConstructTree();
            frame.setTitle("TreeNodeConstructTree");
            frame.setSize(300, 300);
            frame.setVisible(true);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
    }
```

10.5.2 JTable 组件

JTable 表格组件以表格的形式展示数据。JTable 常用的构造方法如下。

(1) JTable():使用默认模型创建 JTable。

(2) JTable(int numRows,int numColumns):创建具有 numRows 行、numColumns 列的 JTable。

(3) JTable(Object[][] rowData,Object[][] columnNames):建立一个显示二维数组数据的表格,其中 rowData 是表格的数据,columnNames 是表格的列名称。

(4) JTable(TableModel dm):使用 TableModel 创建 JTable。

下面演示使用 JTable(Object[][] rowData,Object[][] columnNames)构造的 JTable。

【例 10.15】 使用二维数组构造 JTable。

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;

public class SimpleTable extends JFrame {
    public SimpleTable() {
        String[] columnNames={ "First Name", "Last Name", "Sport",
                "#of Years", "Vegetarian" }; //表头名称
        Object[][] data={
                { "Kathy", "Smith", "Snowboarding", new Integer(5),
                        new Boolean(false) },
                { "John", "Doe", "Rowing", new Integer(3), new Boolean(true) },
                { "Sue", "Black", "Knitting", new Integer(2),
                        new Boolean(false) },
                { "Jane", "White", "Speed reading", new Integer(20),
                        new Boolean(true) },
                { "Joe", "Brown", "Pool", new Integer(10), new Boolean(false) } }; //表数据
```

```
            JTable table=new JTable(data, columnNames);
            JScrollPane scrollPane=new JScrollPane(table);
            add(scrollPane);
        }

        public static void main(String[] args) {
            SimpleTable frame=new SimpleTable();
            frame.setSize(300, 300);
            frame.setVisible(true);
            frame.setTitle("Simple Table");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
    }
```

程序运行结果如图 10-23 所示。

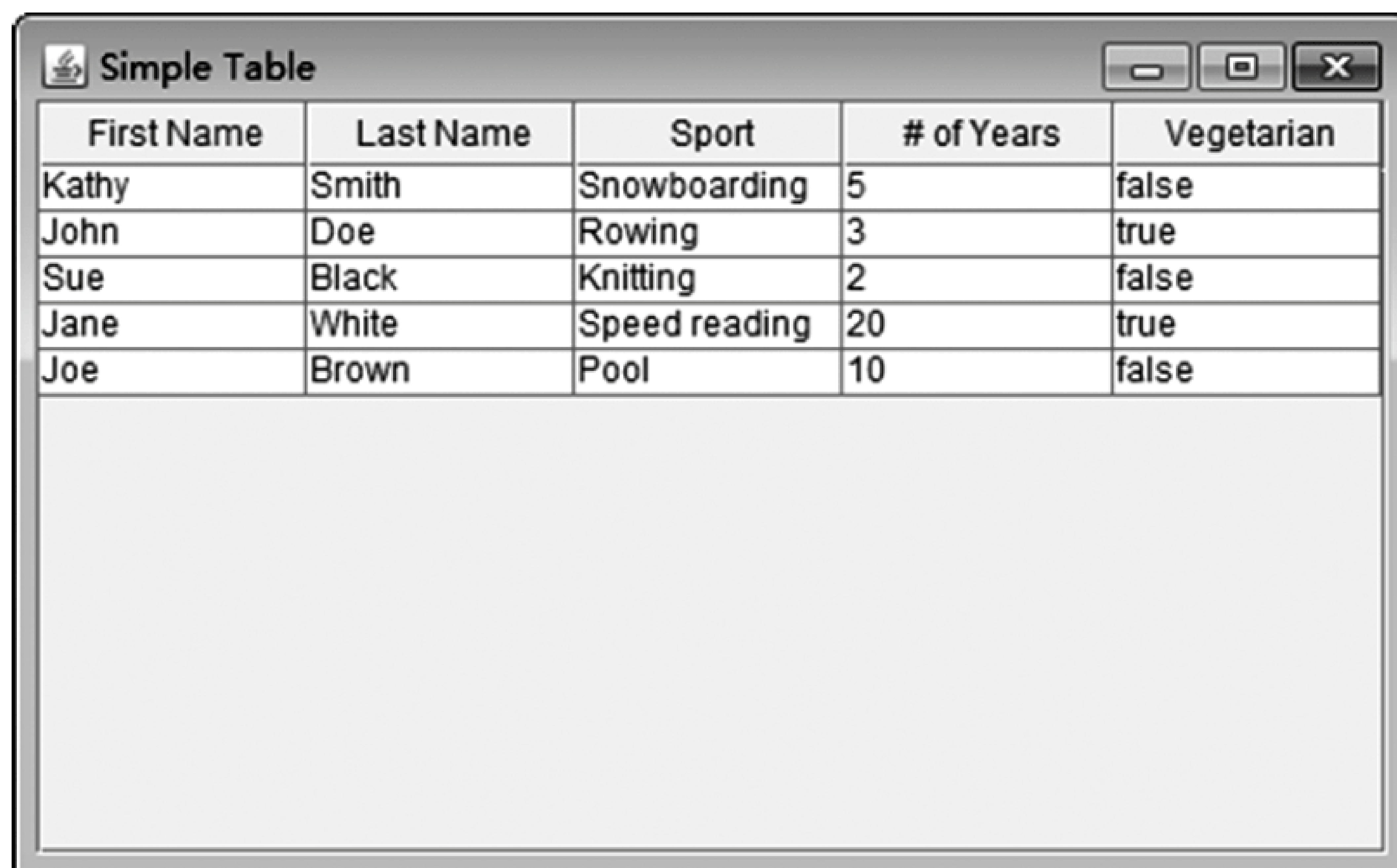


图 10-23　使用二维数组构造 JTable

上例中数组 data 包含了构成表格的数据，columnNames 对应表格的表头部分。

由于 java Swing 采用了 MVC 的设计模式，所以 JTable 只用于视图展示，并不存储数据，真正用来存储和维护数据的是 TableModel 这个接口的实现类。在所有的构造方法中，TableModel 构造 JTable 是较为常用的一种。

TableModel 是一个接口，在这个接口里面定义了若干的方法，这些方法包括存取表格单元的内容、计算表格的列数等的基本存取操作，让设计者可以简单地利用 TableModel 来实现所想要的表格。Swing 提供了 AbstractTableModel 抽象类。该类实现了大部分的 TableModel 方法，让用户可以很有自由地构造自己的表格模式。

【例 10.16】 使用 AbstractTableModel 构造 JTable。

```
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTable;
```

```
public class AbstractTableModelDemo extends JFrame{
public AbstractTableModelDemo() {
    MyTable myTable=new MyTable();
    JTable t=new JTable(myTable);
    JScrollPane s=new JScrollPane(t);
    add(s);
}
public static void main(String args[]) {
    AbstractTableModelDemo frame=new AbstractTableModelDemo();
    frame.setTitle("JTable");
    frame.setSize(300, 300);
    frame.setVisible(true);
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}

import javax.swing.table.AbstractTableModel;
public class MyTable extends AbstractTableModel{
    String[] columnNames={"First Name", "Last Name",  "Sport",  "#of Years",
    "Vegetarian"};                  //表头名称
    Object[][] data={               //表数据
        {"Kathy", "Smith", "Snowboarding", new Integer(5), new Boolean(false)},
        {"John", "Doe",  "Rowing", new Integer(3), new Boolean(true)},
        {"Sue", "Black", "Knitting", new Integer(2), new Boolean(false)},
        {"Jane", "White", "Speed reading", new Integer(20), new Boolean(true)},
        {"Joe", "Brown", "Pool", new Integer(10), new Boolean(false)}
    };

    public Class getColumnClass(int c) {
        return getValueAt(0, c).getClass();
    }

    public int getRowCount() {
        return data.length;
    }

    public int getColumnCount() {
        return columnNames.length;
    }

    public Object getValueAt(int rowIndex, int columnIndex) {
        return data[rowIndex][columnIndex];
    }

}
```

程序运行结果如图 10-24 所示。

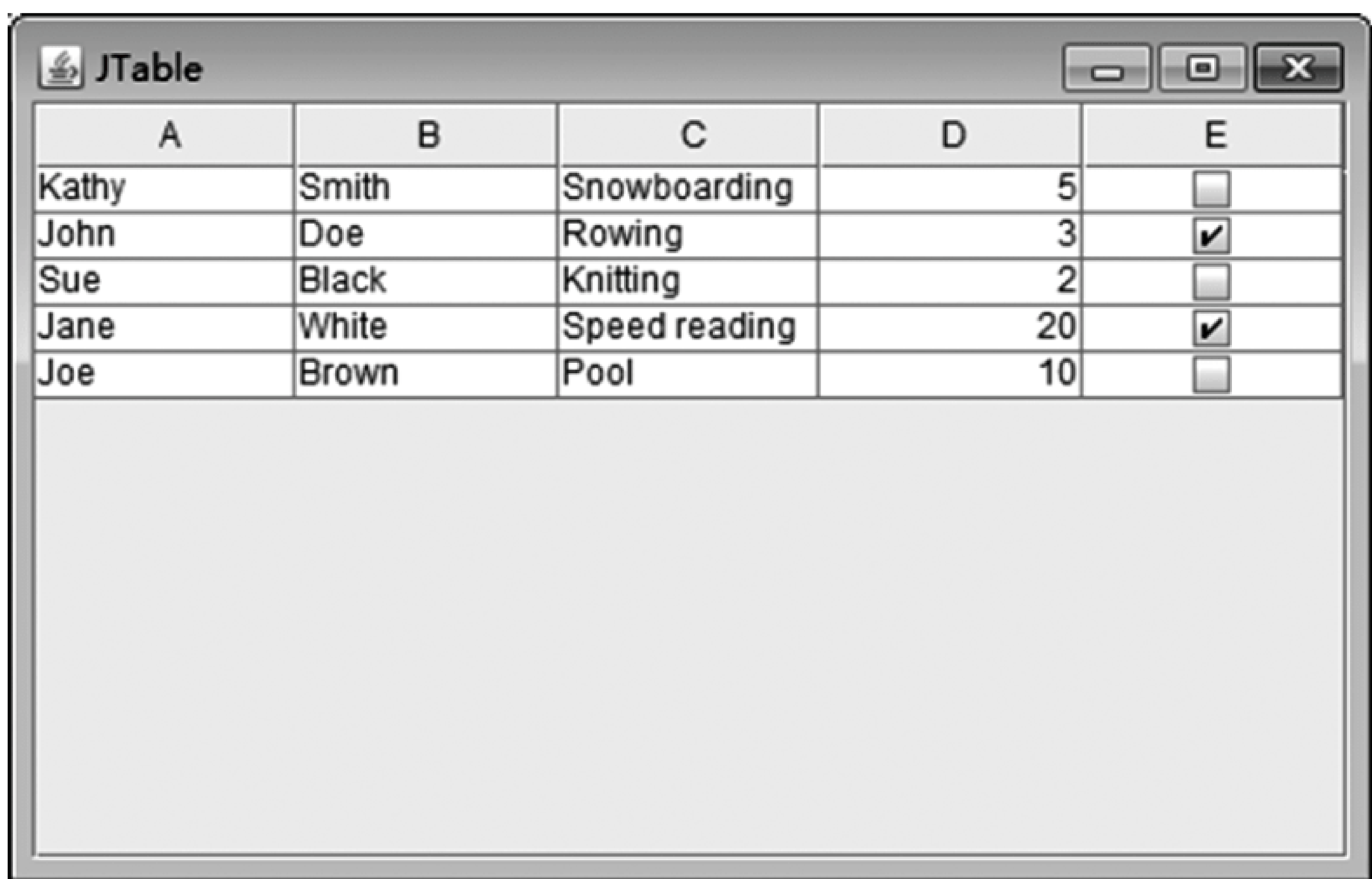


图 10-24　使用 AbstractTableModel 构造 JTable

10.5.3　实用案例

【例 10.17】　动态表格。

本例使用 DefaultTableModel(AbstractTableModel 的子类)实现对表格的增加行列、删除行列的操作。

```
import java.awt.BorderLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Vector;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTable;
import javax.swing.table.DefaultTableModel;
import javax.swing.table.TableColumn;
import javax.swing.table.TableColumnModel;

public class AddAndRemoveCells extends JFrame implements ActionListener {
    private JTable table;
    private DefaultTableModel defaultModel;

    public AddAndRemoveCells() {
        String[] name= { "First Name", "Last Name", "Sport" };       //表头名称
        Object[][] data= {                                            //表数据
                { "Kathy", "Smith", "Snowboarding" }, { "John", "Doe", "Rowing" },
                { "Sue", "Black", "Knitting" },
                { "Jane", "White", "Speed reading" }, { "Joe", "Brown", "Pool" } };
```

```
        defaultModel= new DefaultTableModel(data, name);
        table= new JTable(defaultModel);
        JScrollPane scrollPan= new JScrollPane(table);

        JButton button1= new JButton("增加行");
        JButton button2= new JButton("增加列");
        JButton button3= new JButton("删除行");
        JButton button4= new JButton("删除列");

        JPanel panel= new JPanel();
        panel.add(button1);
        panel.add(button2);
        panel.add(button3);
        panel.add(button4);

        button1.addActionListener(this);
        button2.addActionListener(this);
        button3.addActionListener(this);
        button4.addActionListener(this);

        add(panel, BorderLayout.NORTH);
        add(scrollPan, BorderLayout.CENTER);
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand().equals("增加列"))
            defaultModel.addColumn("新增列");
        if (e.getActionCommand().equals("增加行"))
            defaultModel.addRow(new Vector());
        if (e.getActionCommand().equals("删除列")) {

            int columncount= defaultModel.getColumnCount()- 1;
            if (columncount>= 0) {
                //若 columncount< 0代表已经没有任何列了
                TableColumnModel columnModel= table.getColumnModel();
                TableColumn tableColumn= columnModel.getColumn(columncount);
                columnModel.removeColumn(tableColumn);
                defaultModel.setColumnCount(columncount);
            }
        }
        if (e.getActionCommand().equals("删除行")) {
            //getRowCount 返回行数,rowcount< 0代表已经没有任何行了
            int rowcount= defaultModel.getRowCount()- 1;
            if (rowcount>= 0) {
                defaultModel.removeRow(rowcount);
```

```
                defaultModel.setRowCount(rowcount);
            }
        }
        table.revalidate();
    }

    public static void main(String args[]) {
        AddAndRemoveCells frame=new AddAndRemoveCells();
        frame.setSize(500, 300);
        frame.setVisible(true);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```

程序运行结果如图 10-25 所示。

增加行 增加列 删除行 删除列

First Name	Last Name	Sport	新增列
Kathy	Smith	Snowboarding	
John	Doe	Rowing	
Sue	Black	Knitting	
Jane	White	Speed reading	
Joe	Brown	Pool	

图 10-25 例 10.17 运行结果

10.6 GUI 程序设计实训任务

【任务描述】

编写一个 Swing，模拟实现一个可视化的简单计算器，至少提供包括加法、减法、乘法、除法等基本操作，希望能支持包括正负号、平方根、清零等其他功能。

【任务分析】

该任务的实现需要解决好 3 个问题：①首先要对计算器的按钮、显示栏等进行规划、布局；②对需要进行事件处理和响应的组件设置事件监听器，以获取用户的行为；③在方法 actionPerformed()中逐一判断、分析用户的操作，并根据用户操作(如单击＋)的不同进行事件的响应(如执行加法)，并显示运算结果。

【任务解决】

完整示例代码如下：

```
import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

```
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JTextField;
public class Calculator extends JFrame implements ActionListener {
    private JPanel centerPanel=new JPanel();
    private String x="";
    private String y="";
    private String fh="";
    private double answer;
    private JTextField tfAnswer;

    public Calculator() {
        this.setBackground(Color.lightGray);
        setLayout(new BorderLayout());
        tfAnswer=new JTextField();
        tfAnswer.setHorizontalAlignment(JTextField.RIGHT);
        tfAnswer.setText("0.");
        add(tfAnswer, BorderLayout.NORTH);

        centerPanel.setLayout(new GridLayout(5, 5));
        addButton("7", Color.blue);
        addButton("8", Color.blue);
        addButton("9", Color.blue);
        addButton("/", Color.red);
        addButton("C", Color.red);
        addButton("4", Color.blue);
        addButton("5", Color.blue);
        addButton("6", Color.blue);
        addButton("*", Color.red);
        addButton("^", Color.blue);
        addButton("1", Color.blue);
        addButton("2", Color.blue);
        addButton("3", Color.blue);
        addButton("-", Color.red);
        addButton("sqrt", Color.blue);
        addButton("0", Color.blue);
        addButton("+/-", Color.blue);
        addButton(".", Color.blue);
        addButton("+", Color.red);
        addButton("=", Color.red);
        addButton("sin", Color.red);
        addButton("cos", Color.red);
        addButton("tan", Color.red);
        addButton("ln", Color.red);
```

```
        add(centerPanel, BorderLayout.CENTER);
    }

    public void addButton(String name, Color color) {

        JButton bt=new JButton(name);
        bt.setBackground(Color.white);
        bt.setForeground(color);
        bt.addActionListener(this);
        centerPanel.add(bt);
    }

    //计算功能实现
    public void dengyu(String z) {
        if (z.equals("+"))
            answer=Double.parseDouble(x)+Double.parseDouble(y);
        if (z.equals("-"))
            answer=Double.parseDouble(x)-Double.parseDouble(y);
        if (z.equals("*"))
            answer=Double.parseDouble(x) * Double.parseDouble(y);
        if (z.equals("/"))
            answer=Double.parseDouble(x)/Double.parseDouble(y);
        if (z.equals("^"))
            answer=Math.pow(Double.parseDouble(x), Double.parseDouble(y));
        if (z.equals("sin"))
            answer=Math.sin(Double.parseDouble(x));
        if (z.equals("cos"))
            answer=Math.cos(Double.parseDouble(x));
        if (z.equals("tan"))
            answer=Math.tan(Double.parseDouble(x));
        if (z.equals("ln"))
            answer=Math.log(Double.parseDouble(x));
        x=Double.toString(answer);
        tfAnswer.setText(x);
        y="";
        fh="";
    }

    public void actionPerformed(ActionEvent e) throws IndexOutOfBoundsException {
        if (e.getActionCommand().equals("0")
                || e.getActionCommand().equals("1")
                || e.getActionCommand().equals("2")
                || e.getActionCommand().equals("3")
                || e.getActionCommand().equals("4")
```

```
                || e.getActionCommand().equals("5")
                || e.getActionCommand().equals("6")
                || e.getActionCommand().equals("7")
                || e.getActionCommand().equals("8")
                || e.getActionCommand().equals("9")) {
            if (fh.equals("")) {
                x=x+e.getActionCommand();
                if (x.startsWith("00")) {
                    x=x.substring(1);
                }
                tfAnswer.setText(x);
            } else {
                y=y+e.getActionCommand();
                if (y.startsWith("00")) {
                    y=y.substring(1);
                }
                tfAnswer.setText(y);
            }
        }

        if (e.getActionCommand().equals(".")) {
            if (fh.equals("")) {
                int i=0, j=0;
                for (i=0; i<x.length(); i++)
                    if (x.charAt(i)=='.')
                        j++;
                if (j==0)
                    x=x+".";
                tfAnswer.setText(x);
            } else {
                int i=0, j=0;
                for (i=0; i<y.length(); i++)
                    if (y.charAt(i)=='.')
                        j++;
                if (j==0)
                    y=y+".";
                tfAnswer.setText(y);
            }
        }

        if (e.getActionCommand().equals("C")) {
            x="";
            y="";
            fh="";
            tfAnswer.setText("0.");
```

```
}

if (e.getActionCommand().equals("+/-")) {
    if (fh.equals("")) {
        if (x.substring(0, 1).equals("-"))
            x=x.substring(1);
        else
            x="-"+x;
        tfAnswer.setText(x);
    } else {
        if (y.substring(0, 1).equals("-"))
            y=y.substring(1);
        else
            y="-"+y;
        tfAnswer.setText(y);
    }
}

if (e.getActionCommand().equals("sqrt")) {
    if (fh != "")
        dengyu(fh);
    answer=Math.sqrt(Double.parseDouble(x));
    x=Double.toString(answer);
    tfAnswer.setText(x);
}

if (e.getActionCommand().equals("+")) {
    if (fh != "")
        dengyu(fh);
    fh="+";
}
if (e.getActionCommand().equals("-")) {
    if (fh != "")
        dengyu(fh);
    fh="-";
}
if (e.getActionCommand().equals("*")) {
    if (fh != "")
        dengyu(fh);
    fh="*";
}
if (e.getActionCommand().equals("/")) {
    if (fh != "")
        dengyu(fh);
    fh="/";
```

```
            }
            if (e.getActionCommand().equals("^")) {
                if (fh != "")
                    dengyu(fh);
                fh= "^";
            }
            if (e.getActionCommand().equals("sin")) {
                if (fh != "")
                    dengyu(fh);
                answer= Math.sin(Double.parseDouble(x));
                x= Double.toString(answer);
                tfAnswer.setText(x);
            }
            if (e.getActionCommand().equals("cos")) {
                if (fh != "")
                    dengyu(fh);
                answer= Math.cos(Double.parseDouble(x));
                x= Double.toString(answer);
                tfAnswer.setText(x);
            }
            if (e.getActionCommand().equals("tan")) {
                if (fh != "")
                    dengyu(fh);
                answer= Math.tan(Double.parseDouble(x));
                x= Double.toString(answer);
                tfAnswer.setText(x);
            }
            if (e.getActionCommand().equals("ln")) {
                if (fh != "")
                    dengyu(fh);
                answer= Math.log(Double.parseDouble(x));
                x= Double.toString(answer);
                tfAnswer.setText(x);
            }
            if (e.getActionCommand().equals("= "))
                dengyu(fh);
        }

        public static void main(String args[]) {
            Calculator c= new Calculator();
            c.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            c.setSize(400, 400);
            c.setVisible(true);
        }
    }
```

程序运行结果如图 10-26 所示。

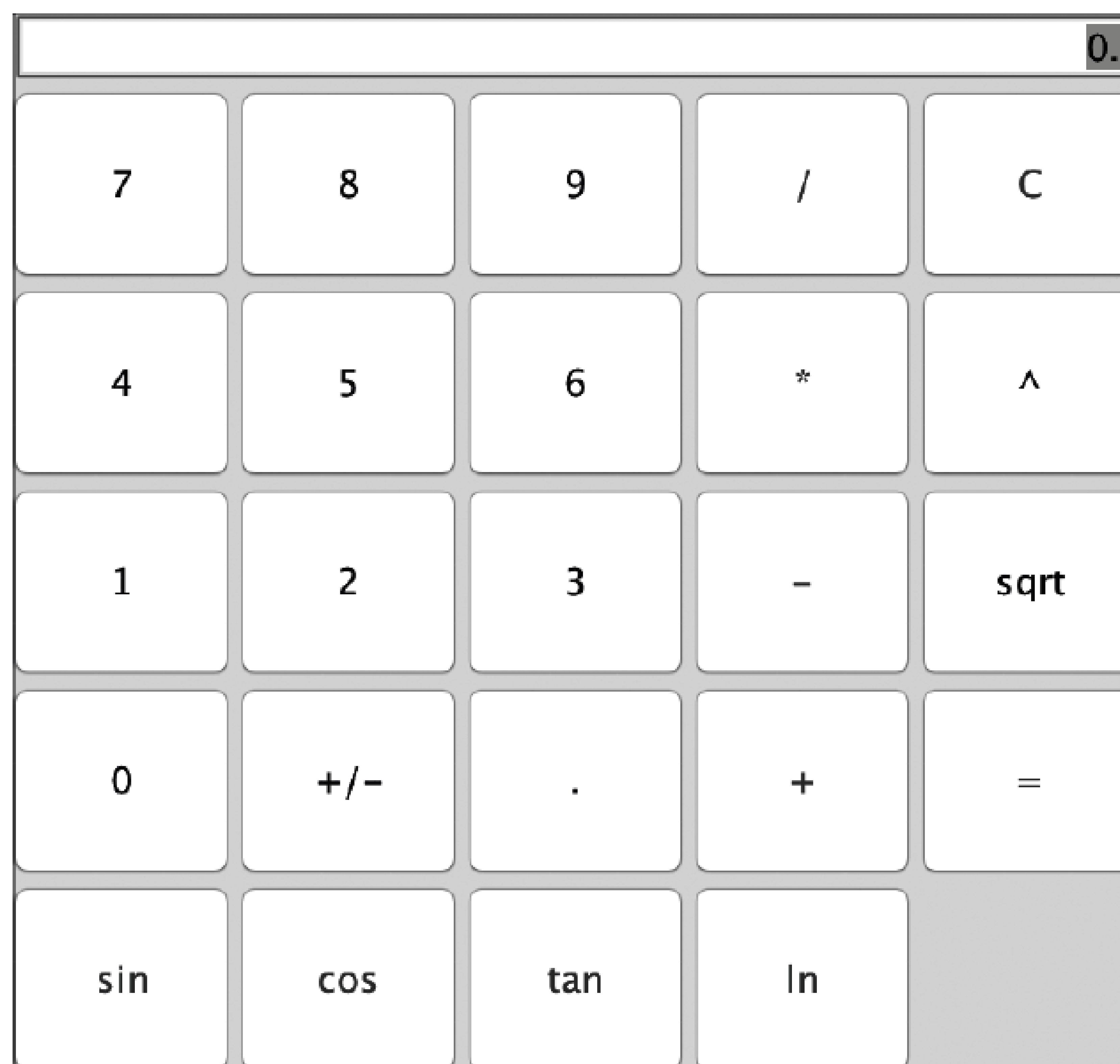


图 10-26　计算器

习题与思考

1. Java 有哪些 GUI 程序设计技术？各有什么区别？

2. 简述 Swing 的类层次结构。

3. 简述 BorderLayout、FlowLayout 和 GridLayout 布局方式的用途。

4. 为什么要使用布局管理器？无布局管理器的布局与有布局管理器的布局二者有何区别？

5. 在 Swing 中如何处理鼠标事件和键盘事件？

6. 编写一个 GUI 程序，包含一个带图标的 JButton 对象。当用户单击这个按钮时，把窗体的标题修改为“单击按钮”。按下按钮和鼠标经过按钮时，JButton 上的图标有不同的效果。

第三篇　网 络 篇

第11章 网络通信

本章学习目标

要使得网络中的两台计算机之间能够相互通信，如果从网络协议的底层去实现也许会比较困难，但是通过一些编程接口和通信模型来实现就简单得多。Java 提供了一系列的编程接口来实现这些功能，本章将主要介绍这些通信接口以及基于这些接口的具体实现，具体内容如下：

(1) URL 类及其 WWW 连接。

(2) 用 ServerSocket 和 Socket 类实现 TCP/IP 客户和服务器。

(3) 用 DatagramPacket 和 DatagramSocket 进行基于 UDP 的网络通信。

(4) 用 MulticastSocket 实现多点广播。

11.1 类 URL 与 URLConnection

通过一个 URL 连接，就可以确定资源的位置，比如网络文件、网页以及网络应用程序等，其中包含了许多具体的语法元素，例如：

```
http://home.netscape.com:80/home/welcome.html
```

这个连接规定使用 http 协议，主机名称为 http://home.netscape.com，端口号为 80。这个 URL 的其他部分/home/welcome.html 则确定了要在这个站点上所要访问的资源。

为了表示 URL，Java 中定义了一个 URL 类，允许程序人员通过它打开特定的 URL 连接，并对连接所对应的资源进行各种读写操作，整个访问过程就像访问本地文件一样方便快捷。

为指向需要访问的 URL 资源，必须先通过以下构造方法初始化一个 URL 对象。

(1) public URL(String spec)：通过一个表示 URL 地址的字符串可以构造一个 URL 对象。

```
URL urlBase=new URL("http://www.cqu.edu.cn/");
```

(2) public URL (URL context, String spec)：通过基于 URL 和相对 URL 构造 URL

对象。

```
URL urlBase=new URL("http://www.cqu.edu.cn/pages/");
URL cquGames=new URL(urlBase, "cqugame.html");
```

(3) public URL(String protocol, String host, String file)。

```
new URL("http", "www.gamelan.com", "/pages/Gamelan.net.html");
```

(4) public URL(String protocol, String host, int port, String file)。

```
URL cqu=new URL ("http", "www.cqu.edu.cn",80, "pages/cqunetwork.html");
```

URL 对象一旦生成,其属性是不能被更改的,但是可以通过类 URL 所提供的以下方法来获取这些属性。

public String getProtocol():获取该 URL 的协议名。

public String getHost():获取该 URL 的主机名。

public int getPort():获取该 URL 的端口号,如果没有设置端口,返回−1。

public String getFile():获取该 URL 的文件名。

public String getRef():获取该 URL 在文件中的相对位置。

在下面的例子中,将生成一个 URL 对象,并获取它的各个属性。

【例 11.1】 URL 对象的属性访问。

```
package code1101;
import java.net.*;
public class ParseURL {
    public static void main(String[] args) throws Exception {
        URL cqu=new URL("http://www.cqu.edu.cn/index.html#top");
        System.out.println("protocol="+cqu.getProtocol()); //协议
        System.out.println("host="+cqu.getHost());          //主机名
        System.out.println("filename="+cqu.getFile());      //文件名
        System.out.println("port="+cqu.getPort());          //端口
        System.out.println("ref="+cqu.getRef());                            //文件内部的一个引用
    }
}
```

程序运行结果

```
protocol=http
host=www.cqu.edu.cn
filename=index.html
port=80
ref=top
```

获得 URL 对象之后,就可以通过 URL 的 openStream()方法读取指定的 WWW 资源,其定义为:

```
InputStream openStream();
```

方法 openStream()与指定的 URL 建立连接并返回 InputStream 类的对象,以便从这一连接中读取数据。

此外,URL 类中还有一个常用的建立远程对象连接的方法:

```
URLConnection openConnection()
```

返回一个 URLConnection 对象,URLConnection 表示应用程序和 URL 之间的通信链接,通过该对象可读、写此 URL 引用的资源,常用的方法如下。

Object getContent():获取此 URL 连接的内容。

String getHeaderField(String name):返回指定的头字段的值。

int getContentLength():返回 content-length 头字段的值。

String getContentType():返回 content-type 头字段的值。

long getLastModified():返回 last-modified 头字段的值。

InputStream getInputStream():返回从此打开的连接读取的输入流。

OutputStream getOutputStream():返回写入到此连接的输出流。

【例 11.2】 通过 URL 和 URLConnection 访问 WWW 资源。

```
package code1101;
import java.util.*;
import java.net.*;
import java.io.*;
public class URLReader {
    public static void main(String[] args) throws Exception {
        try {
            //通过 URL 类建立远程连接,并获取连接内容
            URL obj=new URL("http://www.cqu.edu.cn/");
            BufferedReader in=new BufferedReader(new InputStreamReader
            (obj.openStream()));
            String inputLine;
            while((inputLine=in.readLine()) !=null)
                System.out.println(inputLine);
            in.close();

            //通过 URLConnection 获取响应 Header 信息
            URLConnection conn=obj.openConnection();
            conn.connect();
            System.out.println("获取到的响应长度");
            System.out.println(conn.getContentLength());
            System.out.println("获取到的响应类型 ");
            System.out.println(conn.getContentType());
            //定义 bufferedReader 输入流来读取 URL 的响应
            in=new BufferedReader(new InputStreamReader(conn.getInputStream()));
            String line;
            String result=null;
```

```
                while((line=in.readLine()) != null) {
                    result+=line;
                }
                System.out.println(result);
                in.close()
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
}
```

程序运行结果：

```
<! Doctype Html public "-//W3C//DTD//Html 4.0 Final//EN">
<html>
<head>
…
</body>
</html>
获取到的响应长度
17666
获取到的响应类型
text/html; charset=gb2312
…
```

分析：该例中首先生成一个 URL 对象 cqu，指向重庆大学的主页，然后调用 cqu.openStream()方法生成该 URL 的一个输入流，这是一个字节流，在此基础上进一步通过 InputStreamReader 和 BufferedReader 构造一个带有缓冲功能的字符流，并通过该字符流对象读取该 URL 的 html 内容，进而输出到屏幕上。URLConnection 类也可以用来对由 URL 引用的资源进行读写操作，前提是先通过 connect()方法建立连接，然后再获取响应头信息或响应内容。

实用案例

【例 11.3】 实现一个单线程的资源下载器。

实现一个简单的基于单线程的资源下载器(如图 11-1 所示)，用户可以任意指定待下载资源的链接地址，系统根据该地址判断资源是否存在，如果存在，则将该资源文件下载到本地。

```
package code1101;
public class SingleThreadDown extends JFrame implements ActionListener {
    private final JPanel panel=new JPanel();
    private final JLabel label1=new JLabel("网络资源的单线程下载：");
    private final JLabel label2=new JLabel("网络资源的网址：");
    JButton StartButton=new JButton("单击开始下载");
```

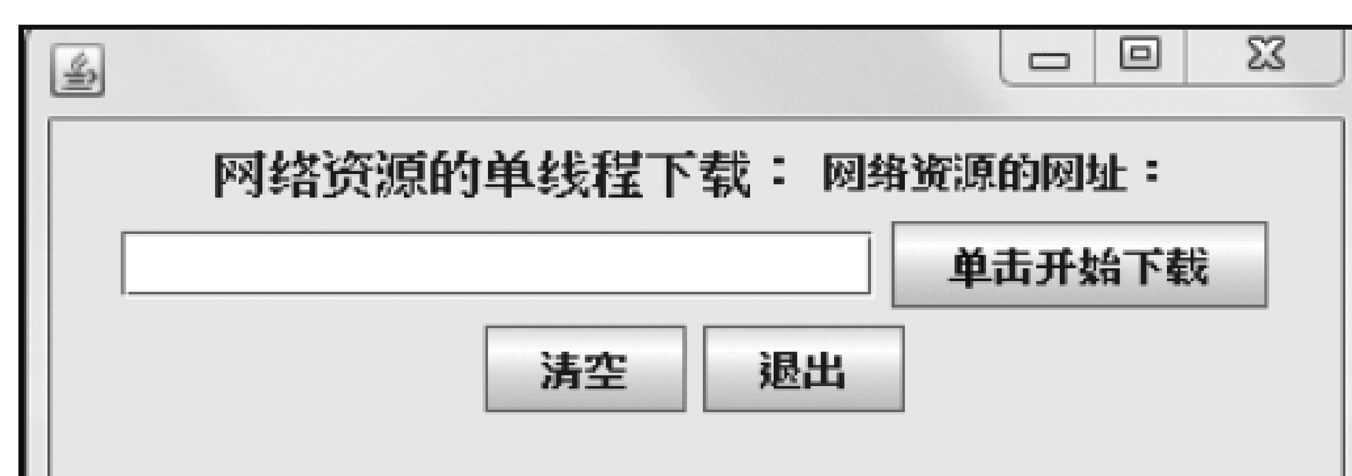


图 11-1　资源下载器基本界面

```
JButton resetButton= new JButton("清空");
JButton exitButton= new JButton("退出");
JTextField urlField= new JTextField(20);

public SingleThreadDown() {
    panel.setLayout(new FlowLayout());
    label1.setFont(new Font("雅黑", Font.BOLD, 15));
    panel.add(label1);
    panel.add(label2);
    panel.add(urlField);
    panel.add(StartButton);
    panel.add(resetButton);
    panel.add(exitButton);
    setContentPane(panel);
    StartButton.addActionListener(this);
    resetButton.addActionListener(this);
    exitButton.addActionListener(this);
    setSize(400, 400);
    setVisible(true);
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

public void download(String address) throws Exception {
    URL url= new URL(address);
    URLConnection urlcon= url.openConnection();
    urlcon.connect();
    InputStream in= urlcon.getInputStream();
    String filePath= url.getFile();
    int pos= filePath.lastIndexOf("/");
    String fileName= filePath.substring(pos+ 1);
    FileOutputStream out= new FileOutputStream("D:\\"+ fileName);
    byte[] bytes= new byte[1024];
    int len= in.read();
    while(len != - 1) {
        out.write(bytes, 0, len);
        len= in.read();
    }
    out.close();
```

```
            in.close();
            JOptionPane.showMessageDialog(this, "下载完毕");
        }

        public void actionPerformed(ActionEvent e) {
            if (e.getSource()==StartButton) {
                if ("".equals(urlField.getText())) {
                    JOptionPane.showMessageDialog(this, "请输入资源地址");
                }
                String url=urlField.getText();
                try {
                    download(url);
                } catch (Exception e1) {
                    JOptionPane.showMessageDialog(this, "资源地址有误,请检查,谢谢!");
                    e1.printStackTrace();
                }
            } else if (e.getSource()==resetButton) {
                urlField.setText("");
            } else {
                System.exit(0);
            }
        }
        public static void main(String[] args) {
            new SingleThreadDown();
        }
    }
```

需要说明的是,为了更加高效地读取远程资源信息,实用的资源下载器(如迅雷等)均为多线程开发,可以根据第 9 章的知识对上述单线程资源下载器进行扩展。

11.2 类 InetAddress

Internet 上的主机通常有两种表示地址的方式:域名和 IP 地址。有时候需要通过域名来查找它对应的 IP 地址,有时候又需要通过 IP 地址来查找主机名。这时候可以利用 java.net 包中的 InetAddress 类来完成任务。

InetAddress 类是 IP 地址封装类,它没有提供可用的构造方法,因此只能利用该类的一些静态方法来获取对象实例,然后再通过这些对象实例来对 IP 地址或主机名进行处理。该类常用的一些方法如下。

pulic static InetAddress getByName(String hostname):在给定主机名的情况下确定主机的 IP 地址。

public static InetAddress getByAddress(byte[] addr):在给定原始 IP 地址的情况下,返回 InetAddress 对象。

public String getHostAddress():获取 IP 地址。

public String getHostName()：获取主机名。

public boolean isReachable(int timeout)：测试是否可以达到该地址。

【例 11.4】 根据指定的域名查找 IP 地址。

```
package code1102;
import java.net.* ;
public class GetIP {
    public static void main(String[] args) {
        try{
            System.out.println("本机 IP 为: "+ InetAddress.getByName(args[0]));
        }catch(UnknownHostException e1){
            e1.printStackTrace();
        }
    }
}
```

执行该应用程序，java GetIP www. cqu. edu. cn 或在 Eclipse 中设置参数如图 11-2 所示，则运行结果为：

```
本机 IP 为: www.cqu.edu.cn/222.178.10.35
```

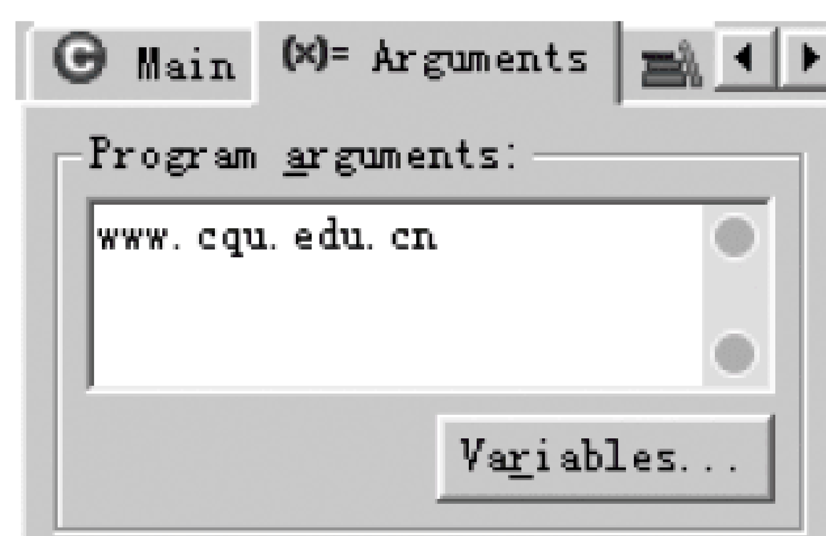


图 11-2　设置参数

实用案例

【例 11.5】 获得指定内网中所有活动 IP。

可以通过循环生成指定内网中的 IP 地址，再利用 InetAddress 类中 isReachable()方法对待测试的 IP 的可达性进行分析。

```
package code1102;
import java.net.InetAddress;
public class TestAllIp {
    public static void main(String[] args) {
        String ip= "172.20.52.";
        for (int i=1; i< 255; i++) {            //循环构建待测试的 IP 地址
            String host= ip+ i;
            new ThreadIP(host).start();
        }
    }
    static class ThreadIP extends Thread {
        String ip= null;
        public ThreadIP(String ip) {
            super();
            this.ip= ip;
        }
        @Override
        public void run() {
            super.run();
```

```
                try {
                    InetAddress ia= InetAddress.getByName(ip);
                    boolean bool= ia.isReachable(1500);                    //判断 IP是否正在被使用
                    if (bool) {
                        System.out.println("主机: "+ ip+ " 可用");
                    }
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }
```

11.3　Socket 通信

Java 中,客户与服务器之间的通信编程一般是基于 Socket 实现的。Socket 是两个实体之间进行通信的有效端点,通过 Socket 可以获得源 IP 地址和源端口、终点 IP 地址和终点端口,并创建一个能被多人使用的分布式应用程序,实现与服务器的双向自由通信。

11.3.1　基于 TCP 协议的 Socket 通信

流式通信协议 TCP 是一种可靠的、基于连接的协议,发送方和接收方所对应的两个 Socket 之间必须建立连接,以便在 TCP 协议的基础上进行通信,当一个 Socket(通常是 Server Socket)等待建立连接时,另一个 Socket 可以要求进行连接,一旦这两个 Socket 连接起来,就可以进行双向数据传输,双方都可以进行发送或接收操作。在 Java 编程语言中,TCP Socket 连接是用 java. net 包中的类实现的。图 11-3 说明了服务器和客户端的通信过程。

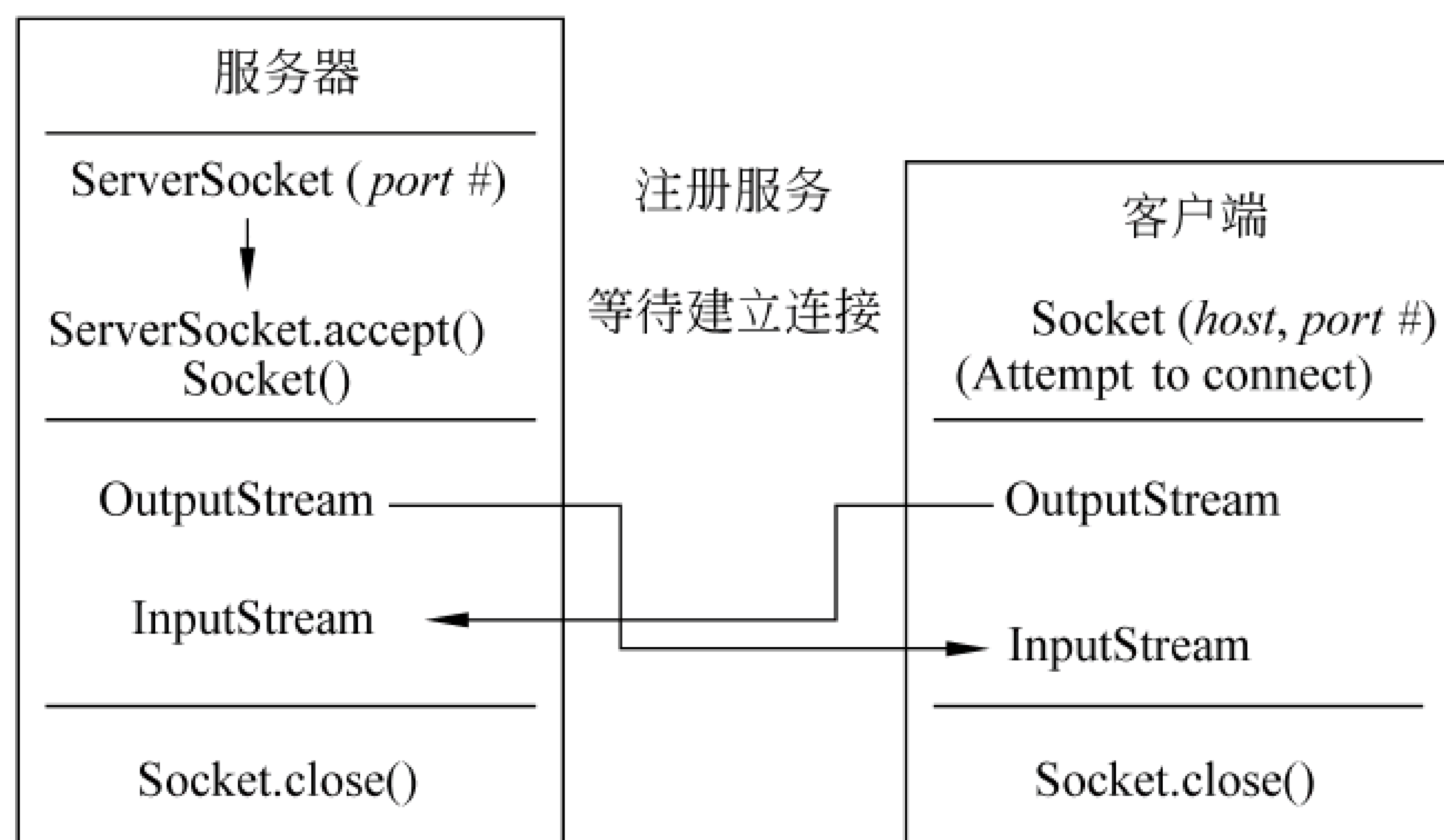


图 11-3　基于 TCP 的 Socket 通信过程

一个完整的 Socket 通信程序通常包括以下几个基本步骤:

(1) 创建 Socket。首先创建 ServerSocket,在客户端创建 Socket 后,连接服务器,在服务器端创建一个和它对应的一个 Socket。

(2) 打开连接到Socket的输入输出流。在客户端和服务器端分别用Socket创建输入流和输出流,把客户端的输入流和服务器的输出流连接起来,客户端的输出流和服务器端的输入流连接起来。

(3) 使用InputStream和OutputStream对Socket进行读写操作。

(4) 使用close()方法关闭Socket连接。通常,程序员主要是针对所要完成的功能在第(3)步进行编程,第(1)、(2)、(4)步对任何程序几乎是一样的。

Java.net包中提供了两个类Socket和ServerSocket,分别用来表示双向连接的客户端和服务端。其构造方法如下:

```
Socket(InetAddress address,int port);
Socket(InetAddress address, int port, Boolean stream);
Socket(String host, int port);
Socket(String host, int port,boolean stream);
ServerSocket(int port);
ServerSocket(int port, int count);
```

其中address、host和port分别是双向连接另一方的IP地址、主机名和端口号,stream指明Socket是流Socket,还是数据报Socket,count则表示服务端所能支持的最大连接数。例如,对于客户端程序,可以通过生成一个Socket对象打开Socket:

```
Socket client;
client=new Socket("Machine name", portNumber);
```

注意:只有给出正确的端口号,才能获得相应的服务。通常,0~1023的端口号为系统所保留,例如,http服务的端口号为80,telnet服务的端口号为23,ftp服务的端口号为21。所以在选择端口号时,应选择一个大于1023的数以防止发生冲突。

对于服务方,通过生成一个ServerSocket对象打开服务器ServerSocket,然后调用方法accept()准备接收客户发来的连接要求,生成一个和客户端对应的Socket。

```
ServerSocket server=null;
try{
    server=new ServerSocket(2000); //2000为端口号
    Socket socket=server.accept();
}
catch(IOException e){
    System.out.println("Error:"+e);
}
```

方法accept()等待客户的请求,直到有一个客户启动并请求连接到相同的端口,然后accept()返回一个对应于客户的Socket。这时,客户方和服务方都建立了用于通信的Socket,接下来就是由各个Socket分别打开各自的输入输出流。

类Socket提供了方法getInputStream()和getOutputStream()来得到对应的输入输出流以进行读写操作,这两个方法分别返回InputStream和OutputStream类对象。为了便于读写数据,可以在返回的输入输出流对象上建立过滤流,如DataInputStream、DataOutputStream类对象,对于文本方式流对象,可以采用InputStreamReader和OutputStreamWriter、

PrintWriter 等处理。例如:

```
PrintStream os=new PrintStream(new BufferedOutputStream(socket.getOutputStream()));
DataInputStream is=new DataInputStream(socket.getInputStream());
PrintWriter out=new printWriter(socket.getOutputStream(),true);
BufferedReader in=new BufferedReader(new InputStreamReader(socket.getInputStream()));
```

使用完 Socket 连接后,应将与 Socket 通信相关的所有资源关闭。

```
os.close();            //关闭输出流
is.close();            //关闭输入流
socket.close();        //关闭 Socket
```

提示:一定要注意关闭的顺序,与 Socket 相关的所有的输入输出流应该首先全部关闭,然后再关闭 Socket。

11.3.2 实用案例

【例 11.6】 单客户端 Socket 通信。

服务器程序如下:

```
package code1103;
import java.net.* ;
import java.io.* ;
public class chatServer {
    public static void main(String args[]) {
        ServerSocket s=null;
        Socket s1;
        String sendString="Hello Net World!";
        OutputStream slout;
        DataOutputStream dos;
        try {
            s=new ServerSocket(5600);                //使用本地 5600 端口提供服务
        } catch (Exception e) {
            e.printStackTrace();
        }
        while(true) {
            try {
                s1=s.accept();              //等待客户端的连接,如果连接成功则返回 Socket 对象
                slout=s1.getOutputStream();
                dos=new DataOutputStream(slout);       //建立输出流
                dos.writeUTF(sendString);
                slout.close();
                s1.close();
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
```

```
        }
    }
```

客户端程序如下：

```
package code1103;
import java.net.* ;
import java.io.* ;
public class chatClient {
    public static void main(String args[]) throws Exception {
        int c;
        Socket s1;
        InputStream s1In;
        DataInputStream dis;
        s1=new Socket("localhost",5600);        //创建客户端 Socket,连接服务器
        s1In=s1.getInputStream();
        dis=new DataInputStream(s1In);          //建立输入流
        String st=new String (dis.readUTF());
        System.out.println(st);
        s1In.close();
        s1.close();
    }
}
```

上述程序只能响应一个客户端程序的连接请求，在实际应用中，服务器一般需要同时响应多个客户请求。因此，ServerSocket 对象的 accept()方法每当有一个连接请求发生时，就会产生一个 Socket 对象，所以只要用此方法反复监听客户请求，就可以为每一个客户生成一个专用的 Socket 对象进行通信。

那么如何管理这么多的 Socket 对象呢？最好的解决办法是将 Socket 对象放置到线程中，这样当每一个 Socket 对象执行任务完成后，只有包含该 Socket 对象的线程会终止，对其他线程没有任何影响。下面是一个可以实现一对多 Socket 通信的例子。

【例 11.7】 多客户端 Socket 通信。

服务器线程程序如下：

```
package code1103;
import java.io.* ;
import java.net.* ;
public class ServerThread extends Thread{
  private Socket s;
  private DataInputStream is;
  private DataOutputStream os;
  //在构造方法中为每个套接字连接输入和输出流
  public ServerThread(Socket socket) throws IOException{
     super();
     s=socket;
     is=new DataInputStream(s.getInputStream());
```

```
            os=new DataOutputStream(s.getOutputStream());
            start(); //启动 run()方法
        }
        //在 run()方法中与客户端通信
        public void run() {
            try{
                String str;
                double result,zhi;
                boolean NotEnd=true;
                while(NotEnd){
                    str=is.readUTF();
                    if(!str.equals("bye")){
                        zhi=Double.parseDouble(str);
                        System.out.println("接收到的值为:"+zhi);
                        result=zhi*zhi;
                        str=Double.toString(result);
                        os.writeUTF(str);
                        os.flush();
                        System.out.println("平方值"+str+"已经发送");
                    }else{
                        NotEnd=false;
                        os.writeUTF("bye");
                        os.flush();
                    }
                }
                is.close();
                os.close();
                s.close();
            }catch(IOException e){
                e.printStackTrace();
            }
        }
    }
```

服务器主程序如下:

```
package code1103;
import java.io.*;
import java.net.*;
public class MultiServer {
    public static void main(String[] args) {
        try{
            System.out.println("等待连接");
            ServerSocket serverSocket=new ServerSocket(5500);
            Socket s=null;
            while(true){
```

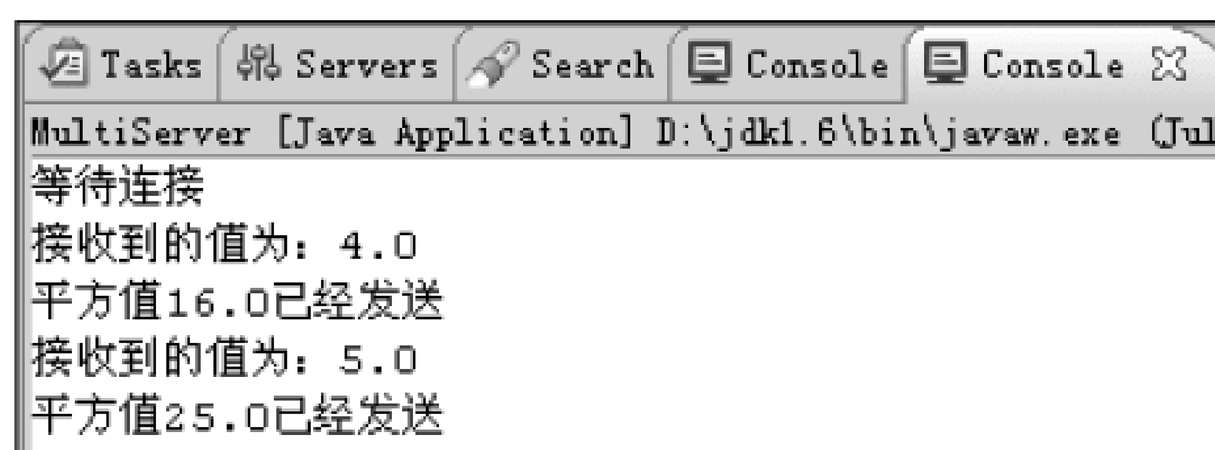
```
            //等待客户端的请求
            s=serverSocket.accept();
            //每次请求都启动一个线程来处理
            new ServerThread(s);
        }
    }catch(IOException e){
        e.printStackTrace();
    }
}
}
```

客户端程序如下：

```
package code1103;
import java.io.*;
import java.net.*;
public class Client {
public static void main(String[] args) {
    try{
      //连接到本机,端口号 5500
      Socket s=new Socket("localhost",5500);
      //将数据输入流连接到 socket 上
      DataInputStream is=new DataInputStream(s.getInputStream());
      //将数据输出流连接到 socket 上
      DataOutputStream os=new DataOutputStream(s.getOutputStream());
      System.out.println("输入待求平方值,输入 bye 结束。");
      String outStr,inStr;
      boolean NotEnd=true;
      BufferedReader buf=new BufferedReader(new InputStreamReader(System.in));
      //反复读用户的数据并计算
      while(NotEnd){
        outStr=buf.readLine();          //读入用户的输入
        os.writeUTF(outStr);            //写到 Socket 中
        os.flush();                     //清空缓冲区,立即发送
        inStr=is.readUTF();             //从 Socket 中读数据
        if(!inStr.equals("bye"))
          System.out.println("返回结果: "+inStr);
        else
            NotEnd=false;
      }
      is.close();
      os.close();
      s.close();
    }catch(IOException e){
        e.printStackTrace();
    }
```

```
    }
  }
```

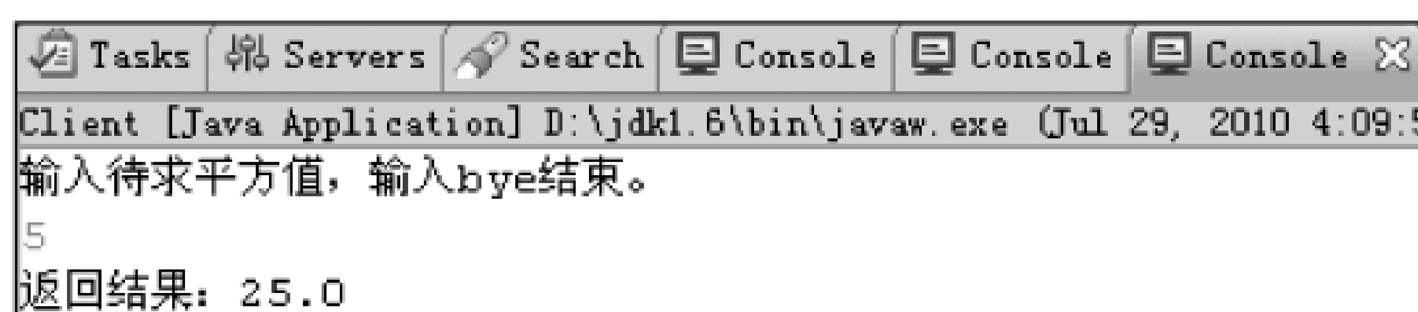
可以先运行服务器主程序,然后多次运行客户端程序,实现多个客户端与同一服务器同时通信的功能。如图 11-4 所示,其中图 11-4(a)为服务器端运行结果,图 11-4(b)为客户端 1 的运行结果,图 11-4(c)为客户端 2 的运行结果。



(a)



(b)



(c)

图 11-4　例 11.7 运行结果

11.3.3　基于 UDP 的网络通信

与 TCP/IP 不同,用户数据报通信协议(User Datagram Protocol,UDP)是一种无连接的协议。每个数据报都是一个独立的信息,包括完整的源地址或目的地址,它在网络上以任何可能的路径传往目的地,因此能否到达目的地,到达目的地的时间以及内容的正确性都是不能保证的。

要区分上述两种协议,一种简单的方法就是把它们比作电话呼叫和邮递信件。电话呼叫保证有一个同步通信,消息按给定次序发送和接收。而对于邮递信件,即使能收到所有的消息,它们的顺序也可能不同。

Java.net 中提供了两个类 DatagramSocket 和 DatagramPacket 用来支持数据报通信。DatagramSocket 用于在程序之间建立传送数据报的通信连接,DatagramPacket 则用来表示一个数据报。DatagramSocket 的构造方法如下:

```
DatagramSocket();
DatagramSocket(int port);
```

其中,port 指明 Socket 所使用的端口号。

用数据报方式编写 Client/Server 程序时,无论在客户方还是服务器方,首先都要建立一个 DatagramSocket 对象,用来接收或发送数据报,然后使用 DatagramPacket 类对象作为传输数据的载体。DatagramPacket 的构造方法如下。

接收时用:

```
DatagramPacket(byte ibuf[],int ilength);
```

发送时用：

```
DatagramPacket(byte ibuf[],int ilength,InetAddress iaddr,int iport);
```

其中的 ibuf 中存放数据报数据，ilength 为数据报中数据的长度，iaddr 和 iport 指明目的地址。

在接收数据前，应用上面的第一种方法先生成一个 DatagramPacket 对象，给出接收数据的缓冲区及其长度。然后调用 DatagramSocket 的方法 receive()等待数据报的到来，receive()一直等待，直到收到一个数据报为止。

```
DatagramPacket packet=new DatagramPacket(buf,256);
socket.receive(packet);
```

发送数据前，也要先生成一个 DatagramPacket 对象，这时要使用上面的第二种构造方法，在给出存放发送数据的缓冲区的同时，还要给出完整的目的地址，包括 IP 地址和端口号。发送数据是通过 DatagramSocket 的方法 send()实现的，send()根据数据报的目的地址来寻径，以传递数据报。

```
DatagramPacket packet=new DatagramPacket(buf,buf.length,address,port);
socket.send(packet);
```

在构造数据报时，要给出 InetAddress 类参数。类 InetAddress 在包 java.net 中定义，用来表示一个 Internet 地址，可以通过它提供的类方法 getByName()从一个表示主机名的字符串获取该主机的 IP 地址，然后再获取相应的地址信息。

11.3.4 实用案例

【例 11.8】 简单的 UDP 通信示例。

客户端程序如下：

```
package code1103;
import java.io.*;
import java.net.*;
import java.util.*;
public class QuoteClient {
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.out.println("Usage:java QuoteClient<hostname>");
            return;
        }
        DatagramSocket socket=new DatagramSocket();      //创建数据报 Socket
        //发送请求
        byte[] buf=new byte[256];
        InetAddress address=InetAddress.getByName(args[0]);          //获取目标地址
        DatagramPacket packet=new DatagramPacket(buf, buf.length, address, 4445);
        //创建发送数据报
```

```
            socket.send(packet);                        //发送
            //获取响应
            packet=new DatagramPacket(buf, buf.length);     //创建接受数据报
            socket.receive(packet);                     //接收数据报
            //显示响应
            String received=new String(packet.getData());
            System.out.println("Quote of the Moment:"+received);
            socket.close();                             //关闭 Socket
        }
}
```

服务端主程序如下：

```
package code1103;
public class QuoteServer {
    public static void main(String args[]) throws java.io.IOException {
        new QuoteServerThread().start();                //启动,创建服务器线程
    }
}
```

服务器线程程序如下：

```
//QuoteServerThread.java
package code1103;
import java.io.*;
import java.net.*;
import java.util.*;
public class QuoteServerThread extends Thread {
    protected DatagramSocket socket=null;
    protected BufferedReader in=null;
    protected boolean moreQuotes=true;
    public QuoteServerThread() throws IOException {
        this("QuoteServerThread");
    }
    public QuoteServerThread(String name) throws IOException {
        super(name);
        socket=new DatagramSocket(4445);        //创建数据报 Socket
        try{ in=new BufferedReader(new FileReader("one.txt"));
        }catch(FileNotFoundException e) {
            System.err.println("Could not open quote file. Serving time instead.");
        }
    }
    public void run() {
        while(moreQuotes) {
            try{
                byte[] buf=new byte[256];
                DatagramPacket packet=new DatagramPacket(buf,buf.length);
```

```
                    //生成数据报,准备接收
                    socket.receive(packet);          //接收
                    String dString=null;
                    if(in==null)
                        dString=new Date().toString();
                    else
                        dString=getNextQuotes();
                    buf=dString.getBytes();
                    //发送响应给客户端,使用收到的数据报的 address 和 port
                    InetAddress address=packet.getAddress();
                    int port=packet.getPort();
                    packet=new DatagramPacket(buf,buf.length,address,port);
                                                     //创建发送数据报
                    socket.send(packet);             //发送
                }catch(IOException e) {
                    e.printStackTrace();
                    moreQuotes=false;
                }
            }
            socket.close();
    }
    //获取下一个需要发送的数据内容
    protected String getNextQuotes(){
      String returnValue=null;
        try {
          if((returnValue=in.readLine())==null) {
          in.close();
          moreQuotes=false;
          returnValue="No more quotes.Goodbye.";
          }
        }catch(IOException e) {
          returnValue="IOException occurred in server";
         }
        return returnValue;
      }
    }
```

从例子中可以看出,编写数据报通信程序还是比较简单的,其重点是掌握 DatagramPacket 和 DatagramSocket 的使用,包括如何创建接收数据报和接收数据报 Socket,以及发送数据报和发送数据报 Socket。创建数据报通信程序,不需要在客户端和服务器端之间建立连接。

11.3.5 基于 MulticastSocket 实现多点广播

DatagramSocket 只允许数据报发送给指定的目标地址,而 MulticastSocket 可以将数据报以广播方式发送到数量不等的多个客户端。

若要使用多点广播时，则需要让一个数据报标有一组目标主机地址，当数据报发出后，整个组的所有主机都能收到该数据报。IP 多点广播实现了将单一信息发送到多个接收者的广播，如图 11-5 所示，其思想是设置一组特殊网络地址作为多点广播地址，每一个多点广播地址都被视为一个组，当客户端需要发送、接收广播信息时，加入到该组即可。IP 协议为多点广播提供了这批特殊的 IP 地址，这些 IP 地址的范围是 224.0.0.0～239.255.255.255。

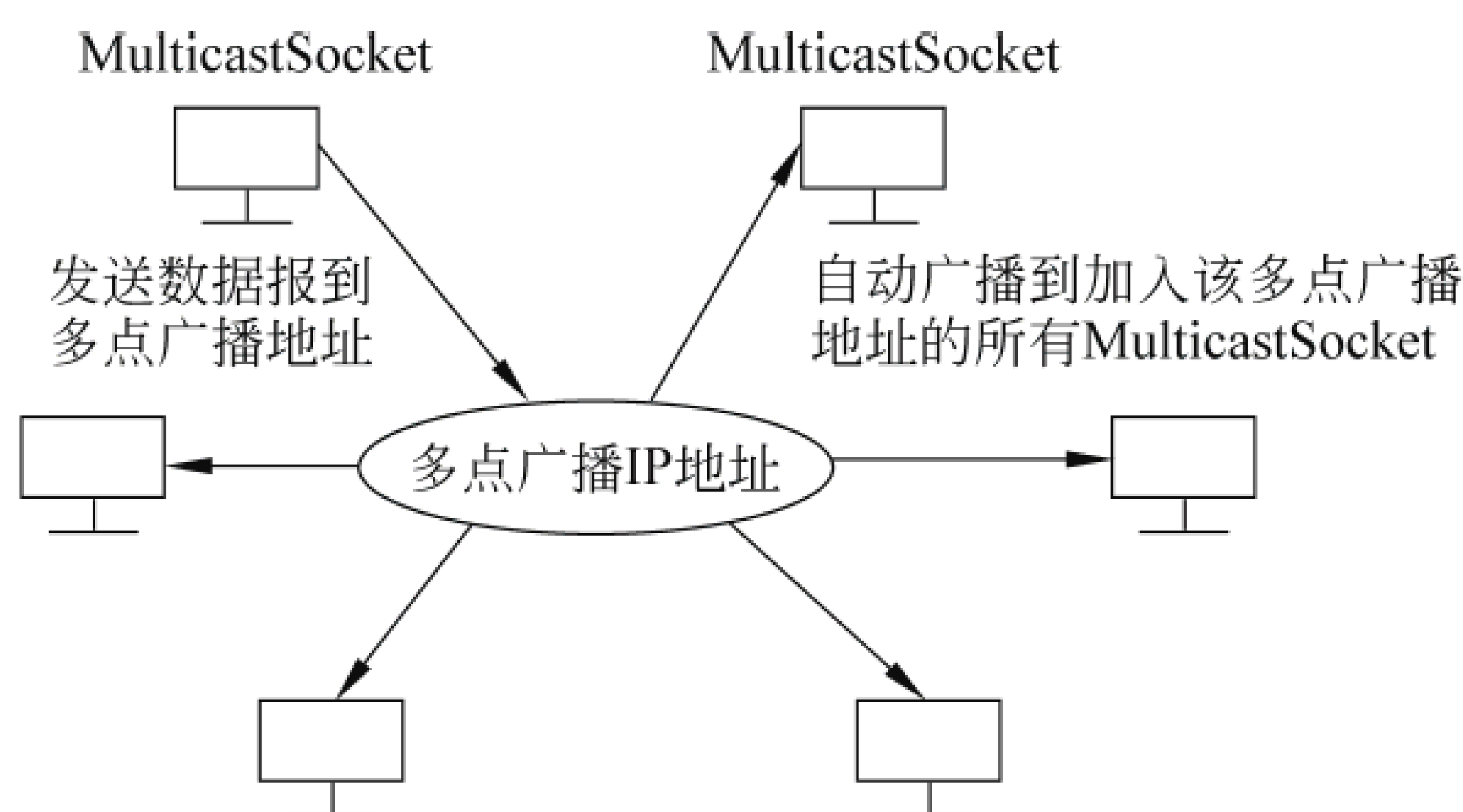


图 11-5　多点广播原理示意图

通过 Java 实现多点广播时，MulticastSocket 类是实现这一功能的关键，当 MulticastSocket 把一个 DatagramPacket 发送到多点广播 IP 地址，该数据报将被自动广播到加入该地址的所有 MulticastSocket。MulticastSocket 类既可以将数据报发送到多点广播地址，也可以接收其他主机的广播信息。

MulticastSocket 类似于 DatagramSocket，要发送一个数据报时，可使用随机端口创建 MulticastSocket，也可以在指定端口来创建 MulticastSocket，对应的主要构造方法如下。

public MulticastSocket()：使用本机地址、随机端口创建 MulticastSocket 对象。

public MulticastSocket(int portNumber)：使用本机地址、指定端口创建 MulticastSocket 对象。

public MulticastSocket(SocketAddress bindaddr)：使用本机指定 IP 地址、指定端口创建 MulticastSocket 对象。

如果创建仅用于发送数据报的 MulticastSocket 对象，则使用默认地址、随机端口即可。但是，如果创建接收用的 MulticastSocket 对象，则该 MulticastSocket 对象必须具有指定端口，否则发送方无法确定发送数据报的目标端口。

创建一个 MulticastSocket 对象后，还需要将该 MulticastSocket 加入到指定的多点广播地址，MulticastSocket 使用 joinGroup()方法来加入指定组，使用 leaveGroup()方法脱离一个组。

```
void joinGroup(InetAddress multicastAddr)
void leaveGroup(InetAddress multicastAddr)
```

MulticastSocket 用于发送、接收数据报的方法与 DatagramSocket 的方法完全一样。但是，MulticastSocket 比 DatagramSocket 多一个 setTimeToLive(int ttl)方法，该方法用于设置在此 MulticastSocket 上发出的多播数据包的默认生存时间，以便控制多播的范围。ttl 必须在 0～255 范围内，否则将抛出 IllegalArgumentException。在默认情况下，该 ttl 的值

为1。

【例11.9】 一个多点广播简单示例。

服务器端程序如下：

```
package code1103;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
public class UDPMulticastServer {
    final static int RECEIVE_LENGTH=1024;
    static String multicastHost="224.0.0.1";
    static int localPort=9998;
    public static void main(String[] args) throws Exception {
        InetAddress receiveAddress=InetAddress.getByName(multicastHost);
        if(!receiveAddress.isMulticastAddress()) {        //测试是否为多播地址
            throw new Exception("请使用多播地址");
        }
        int port=localPort;
        MulticastSocket receiveMulticast=new MulticastSocket(port);
        receiveMulticast.joinGroup(receiveAddress);
        DatagramPacket dp=new DatagramPacket(new byte[RECEIVE_LENGTH],
                RECEIVE_LENGTH);
        receiveMulticast.receive(dp);
        System.out.println(new String(dp.getData()).trim());
        receiveMulticast.close();
    }
}
```

客户端程序如下：

```
package code1103;
import java.net.DatagramPacket;
import java.net.InetAddress;
import java.net.MulticastSocket;
public class UDPMulticastClient {
    static String destAddressStr="224.0.0.1";
    static int destPortInt=9998;
    static int TTLTime=4;
    public static void main(String[] args) throws Exception {
        InetAddress destAddress=InetAddress.getByName(destAddressStr);
        if(!destAddress.isMulticastAddress()) {        //检测该地址是否是多播地址
            throw new Exception("地址不是多播地址");
        }
        int destPort=destPortInt;
        int TTL=TTLTime;
        MulticastSocket multiSocket=new MulticastSocket();
```

```
            multiSocket.setTimeToLive(TTL);
            byte[] sendMSG="Hello".getBytes();
            DatagramPacket dp=new DatagramPacket(sendMSG, sendMSG.length,
                    destAddress, destPort);
            multiSocket.send(dp);
            multiSocket.close();
        }
    }
```

11.4 网络通信实训任务

【任务描述】

使用 Java 访问某些 URL 时,如果该 URL 需要身份验证(即登录),那么就不能直接访问,请编码实现网页模拟登录功能。

【任务分析】

使用 Java 模拟登录的基本思想是:首先用合法的用户名和密码向 Web 服务器提出登录请求,如果请求获得授权,记录下返回的 Cookie 信息,在下次发起请求时将该 Cookie 发送过去用以表明身份,这样就能够访问带有权限的 URL 了。

【任务解决】

```
package code1104;
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.URL;
public class Login {
    public static void main(String[] args) {
        try {
            //验证登录的网页 URL
            URL url=new URL("http://localhost:8088/webapp/action.jsp");
            //打开该网页的 URL 连接
            HttpURLConnection conn=(HttpURLConnection) url.openConnection();
            //设置该网页连接可以向服务器发送信息
            conn.setDoOutput(true);
            //设置请求的提交方式
            conn.setRequestMethod("POST");
            //向服务器发送包含用户名称和密码的登录认证查询串
            conn.getOutputStream().write("username=a&password=a".getBytes());
            conn.getOutputStream().flush();
            conn.getOutputStream().close();
            //获取网页的标头信息中的 Cookie 字符串
            String cookieVal=conn.getHeaderField("Set-Cookie");
            //连接服务器
```

```
            conn.connect();
            conn.disconnect();

            //主网页的 URL
            url=new URL("http://localhost:8088/webapp/index.jsp");
            //打开该网页的 URL 连接
            conn=(HttpURLConnection) url.openConnection();
            //设置当前连接的 Cookie 信息和上次连接的 Cookie 信息一致,维持同一会话状态
            conn.setRequestProperty("Cookie", cookieVal);
            //获取服务器的主页返回结果
            BufferedReader rd=new BufferedReader(new InputStreamReader(conn.getInputStream()));
            //输出主页结果
            String line;
            while((line=rd.readLine()) !=null) {
                System.out.println(line);
            }
            rd.close();
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }
}
```

习题与思考

1. 基于 TCP 与 UDP 协议的通信有什么区别?

2. 创建一个 URL 对象,并通过它读取指定的 WWW 资源。

3. 给定服务器 templates 目录中的文件 FileServer.java,编程实现以下功能:使服务器接收来自客户的文件名字符串,试图打开该文件并将文件内容通过 Socket 传回到客户。

4. 如何找出连接到服务器的客户端的 IP 地址?

5. 下列说法错误的是(　　)。

　A. TCP 是面向连接的协议,而 UDP 是无连接的协议

　B. 数据报传输是可靠的,可以保证包按顺序到达

　C. URL 代表统一资源定位符,通过它可以确定资源的位置

　D. Socket 和 ServerSocket 分别表示连接的 Client 端和 Server 端

第12章 JSP与Servlet技术

本章学习目标

基于Java的Web服务器端编程主要涉及JSP和Servlet技术,JSP与Servlet之间的交互为动态网页的开发提供了优秀的解决方案。本章主要讲述JSP与Servlet技术的基本内容和使用方法。通过本章的学习,应该重点掌握以下主要内容:

(1) JSP开发技术。

(2) Servlet开发技术。

(3) JSP与Servlet的混合使用。

12.1 为什么使用JSP

网页是网络应用中最重要的一种形式,但是一般的网页是静态的。JSP技术可以实现网页的交互性、自动更新、因时因人而变的动态网页特性。JSP页面由HTML代码和嵌入其中的Java代码所组成。服务器在接到客户端请求页面后,对这些Java代码进行处理,然后将生成的HTML页面返回给客户端的浏览器。

【例12.1】 编写一个JSP页面,使之能够显示一个数据表格,表格中的数据会根据行数进行变化。

```
<%@ page language="java" contentType="text/html; charset=gb2312" pageEncoding="gb2312"%>
<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>第一个 JSP 页面</title>
</head>
<body>
<table>
    <tr>
        <td>id</td>
        <td>姓名</td>
```

```
    </tr>
    <%
        String color1="99ccff";
        String color2="88cc33";
        for (int i=1; i<=10; i++) {
            String color="";
            if (i%2==0) {
                color=color1;
            } else {
                color=color2;
            }
            out.println("<tr bgcolor="+color+">");
            out.println("<td>"+i+"</td>");
            out.println("<td>姓名"+i+"</td>");
            out.println("</tr>");
        }
    %>
</table>
</body>
</html>
```

程序可能的运行结果如图 12-1 所示。



图 12-1　例 12.1 程序可能的运行结果

编写一个 JSP 页面涉及以下 3 个关键内容：

(1) 了解 JSP 页面的构成，参见 12.2.2 节。

(2) 定义 JSP 页面的指令、脚本元素，参见 12.2.2 节。

(3) 使用各种 JSP 内建对象，参见 12.2.3 节。

12.2 JSP 技术

12.2.1 JSP 工作原理

JSP 源文件是由安装在 Web 服务器上的 JSP 引擎编译执行的。例如,Tomcat 就是一种 JSP 引擎。JSP 引擎把来自客户端的请求传递给 JSP 源文件,然后 JSP 引擎再把对它的响应从 JSP 源文件传递给客户端。客户端请求和响应的过程如图 12-2 所示。

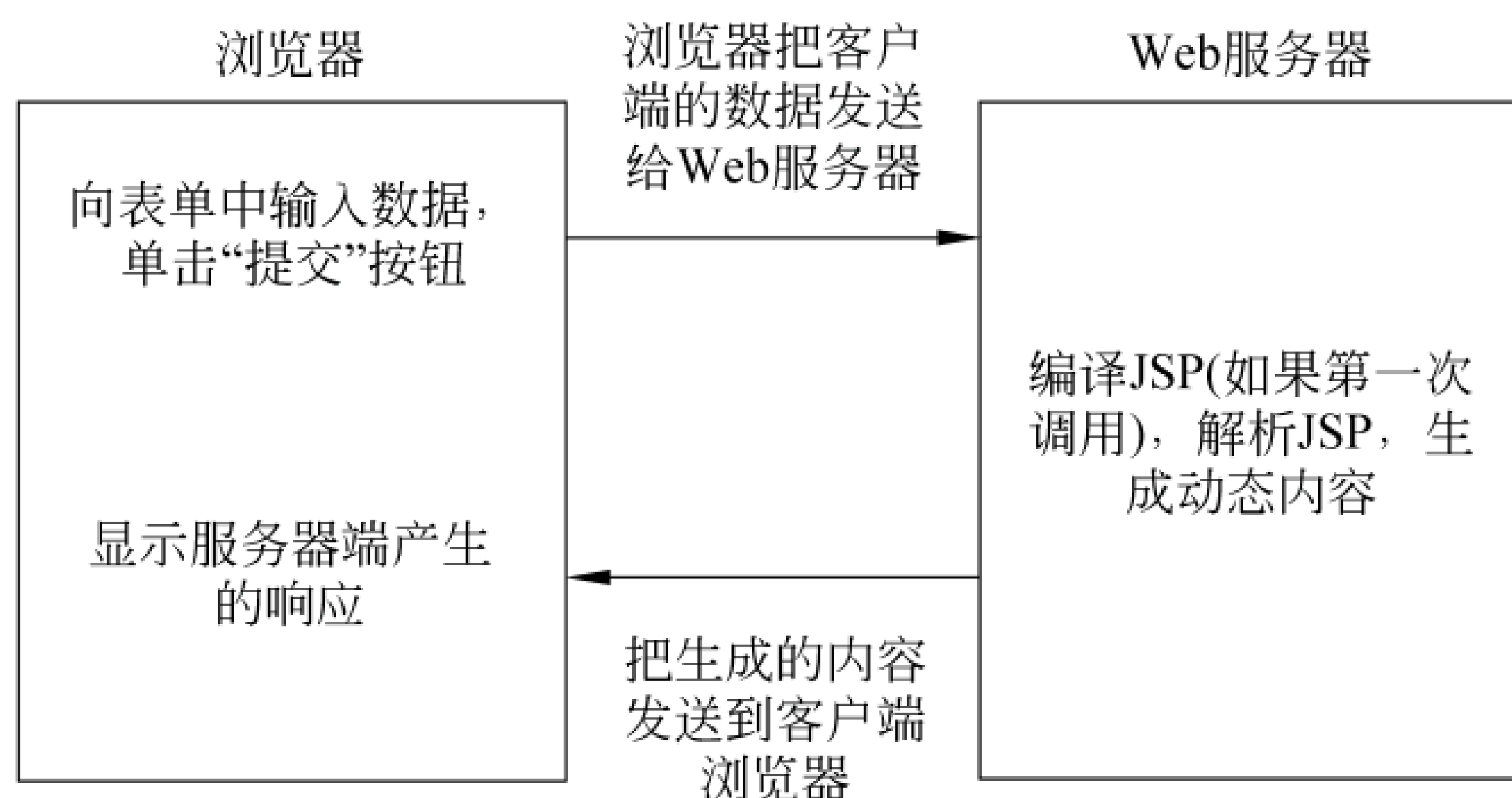


图 12-2 客户端请求和响应的过程

【例 12.2】 编写一个简单的 HTML 网页(JSPParameterDemo.html),该网页中包含一个表单,这个表单可以接收用户输入的数字,然后把这个数字发送到 Web 服务器,服务器根据接收的参数来决定显示内容重复的次数。

JSPParameterDemo.html 网页的代码如下:

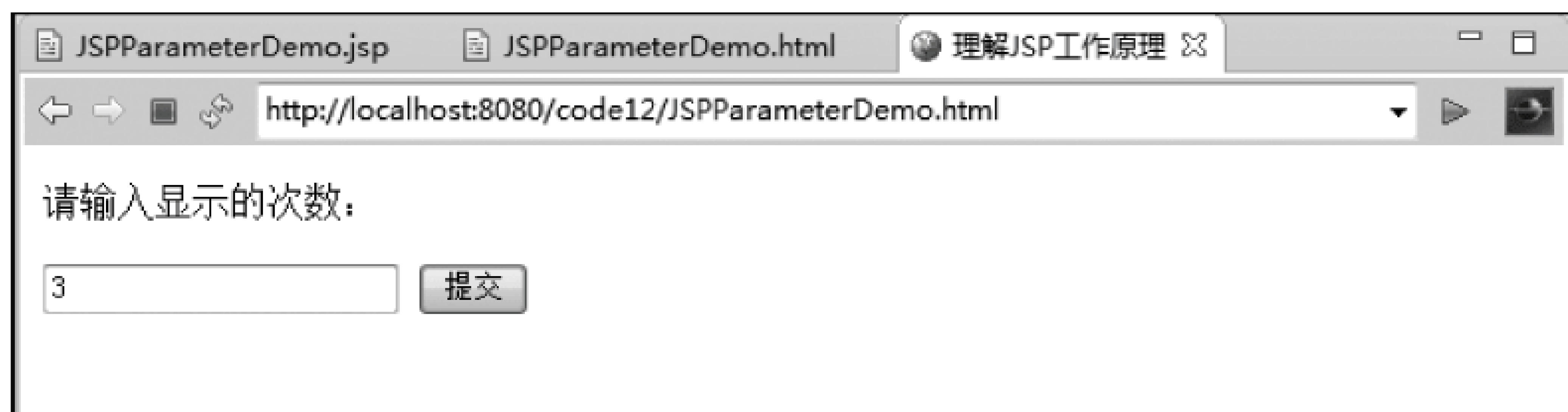
```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>理解 JSP 工作原理</title>
</head>
<body>
<p>请输入显示的次数:</p>
<form action="JSPParameterDemo.jsp" method="get">
    <input type="text" name="times" />
    <input type="submit" value="提交" /></form>
</body>
</html>
```

处理请求的 JSP 页面代码如下:

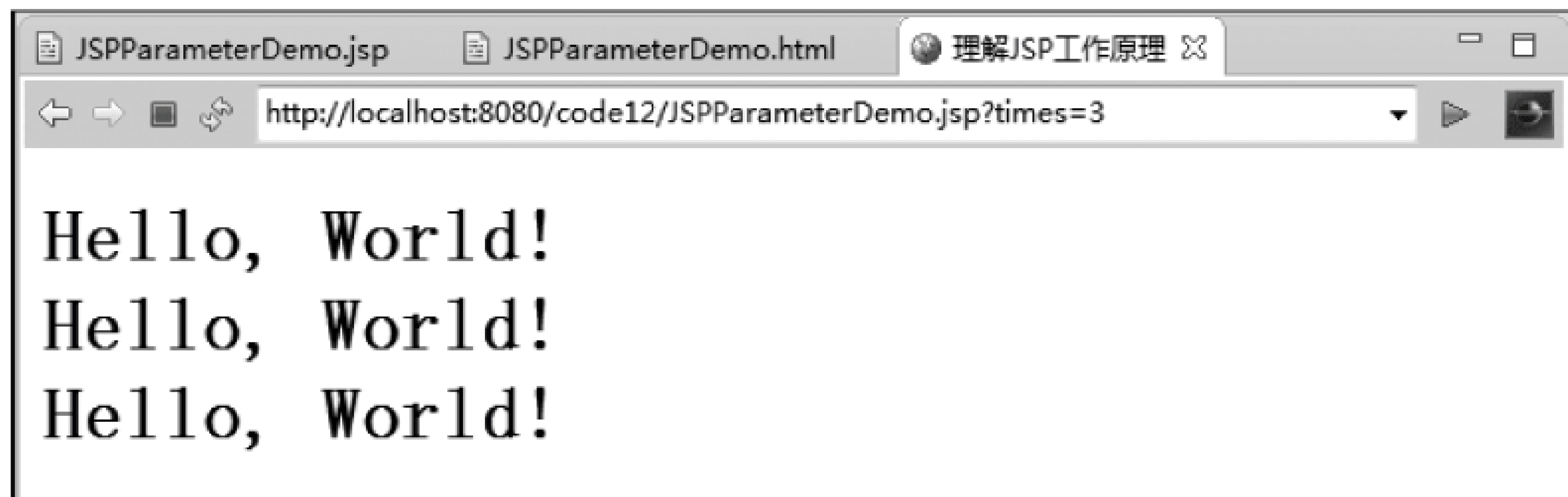
```
<%@ page language="java" contentType="text/html; charset=gb2312"
    pageEncoding="gb2312"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
```

```
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>理解 JSP 工作原理</title>
</head>
<body>
<h1>
<%
    int times=Integer.parseInt(request.getParameter("times"));
    for (int i=0; i<times; i++) {
        out.println("Hello, World!");
        out.println("<br/>");
    }
%>
</h1>
</body>
</html>
```

程序可能的运行结果如图 12-3 所示。



(a)



(b)

图 12-3　例 12.2 程序可能的运行结果

分析：在 JSPParameterDemo.html 网页中，当用户输入数字后单击“提交”按钮，浏览器就会把用户的数据和请求发送给 JSPParameterDemo.jsp 页面，该 JSP 页面会被执行，其生成的内容是 HTML 代码，并传给客户端浏览器进行显示。

JSP 页面生成 HTML 代码如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>理解 JSP 工作原理</title>
```

```
</head>
<body>
<h1>
Hello, World!
<br/>
Hello, World!
<br/>
Hello, World!
<br/>

</h1>
</body>
</html>
```

将上述 HTML 代码与 JSP 页面代码进行比较,可以看到 JSP 页面中的 Scriptlets 代码被替换成了 HTML 网页中的内容。JSP 引擎在第一次调用 JSP 页面时,会自动编译 JSP 页面,然后这个 JSP 就驻留内存,因此第一次调用 JSP 时总会有一定的延时,在接下来的调用中就不会再有延时了。

12.2.2 JSP 的构成

JSP 文件主要由模板元素、指令元素、动作元素、脚本元素、声明、表达式、Scriptlets 和 JSP 内建对象组成。JSP 程序可以使用编辑 HTML 的工具进行编辑,编辑完成后保存成 *.jsp 文件即可。

【例 12.3】 一个典型的 JSP 文件。

```
<%@ page language="java" contentType="text/html; charset=gb2312" pageEncoding="gb2312"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<!--这是一个典型的 JSP 文件,当前访问时间是:<%=(new java.util.Date()).toString()%>-->
<%! String getDate() {
        return new java.util.Date().toString();
   }%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>一个典型的 JSP</title>
</head>
<body>
<%@ include file="header.jsp"%>
<hr />
当前时间是:<%=getDate()%>
<jsp:include page="copy.jsp" flush="true" />
</body>
</html>
```

从上述 JSP 文件中,可以总结出 5 类元素:模板元素、注释、脚本元素、指令元素和动作

元素。

下面分别简要介绍这些元素。

1. 模板元素

模板元素是指JSP的静态HTML内容。这些模板元素可以说是网页的框架，它影响页面的结构和美观程度。当客户端请求JSP页面时，它会把这些模板元素一字不变地发送到客户端。

2. 注释

JSP中的注释有3种：HTML注释、隐藏注释和Scriptlets中的注释。

(1) HTML注释：在客户端显示一个注释。

JSP语法：

```
<!--commentInfo [<%=expression%>] -->
```

这种注释和HTML中的很像，唯一不同之处是，可以在这个注释中使用表达式，表达式的结果是不定的，由页面来决定。

如在例12.3中：

```
<!--这一个典型的JSP,当前访问时间是:<%=(new java.util.Date()).toString()%>-->
```

在客户端的HTML源代码中显示为：

```
<!--这是一个典型的JSP文件,当前访问时间是:Sun Jul 04 11:03:40 CST 2010 -->
```

(2) 隐藏注释：写在JSP文件中，但不发送给客户端。

JSP语法：

```
<%--commentInfo --%>
```

用隐藏注释标记的字符会在JSP编译时被忽略。

(3) Scriptlets中的注释：由于Scriptlets包含的是Java代码，所以Java中的注释规范在Scriptlets中也能使用，常用的Java注释包括使用//表示单行注释，使用/*…*/来表示多行注释。

3. 脚本元素

JSP脚本元素是JSP代码中使用最频繁的元素，它通常是用Java写的脚本代码。脚本元素主要包括声明(declaration)、表达式(expression)和Scriptlets。

(1) 声明：就是在JSP程序中声明合法的变量和方法。如例12.3中的声明：

```
<%!
    String getDate() {
            return new java.util.Date().toString();
    }%>
```

(2) 表达式：就是位于<%=和%>之间的代码，它在JSP请求处理阶段计算其值，所得的结果转换成字符串并与模板数据组合在一起。表达式在页面的位置，也就是该表达式计算结果所在的位置。如例12.3中的表达式：

```
<%=getDate()%>
```

(3) Scriptlets:位于<%和%>之间,它是一段可以在处理请求时执行的 Java 代码。它可以产生输入,并将输出结果发送到客户端的输出流里,也可以是一些流程控制语句。

4. 指令元素

指令用于从 JSP 发送一个消息到容器上。它用来设置全局变量,声明类、方法和输出内容的类型等。它们并不向客户端产生任何输出,所有的指令都在 JSP 整个文件范围内有效。指令元素使用以下的格式:

```
<%@ directivename attribute="value", attribute="value"%>
```

JSP 中常见的指令元素有以下两种。

(1) 页面(page)指令:页面指令用来定义 JSP 文件中的全局属性。如例 12.3 中的页面指令:

```
<%@ page language="java" contentType="text/html; charset=gb2312"
    pageEncoding="gb2312"%>
```

该语句定义了 JSP 文件所使用的脚本语言是 Java,响应的内容类型是 text/html,使用的字符编码集是 gb2312,JSP 页面的字符编码集是 gb2312。

(2) include 指令:include 指令通知容器将指定位置上的资源内容包含到当前 JSP 页面中。被包含的文件内容可以被 JSP 解析,并且一经编译,内容不可变,如果要改变被包含文件的内容,必须重新编译 JSP 文件。如例 12.3 中的 include 指令:

```
<%@ include file="header.jsp"%>
```

5. 动作元素

与指令元素不同,动作元素在请求处理阶段起作用。JSP 动作元素使用 XML 语法写成,它采用以下两种格式中的一种:

```
<prefix:tag attribute=value attribute-list…/>
```

或者

```
<prefix:tag attribute=value attribute-list…>
…
</prefix:tag>
```

容器在处理 JSP 时,每遇到动作元素,都根据它的标记进行特殊的处理。JSP 规范定义了一系列的标准动作,它们用 jsp 作为前缀。常见的动作元素有<jsp:include>、<jsp:forward>、<jsp:useBean>、<jsp:setProperty>和<jsp:getProperty>等。

(1) <jsp:include>:该操作允许在请求时间内在现有 JSP 页面中包含静态或者动态的资源。其语法格式如下:

```
<jsp:include page="fileName" flush="true"/>
```

(2) <jsp:forward>:该操作允许将请求转发给另一个 JSP、Servlet 或者静态资源文件。每当遇到此操作时,就停止执行当前的 JSP,转而执行被转发的资源。其语法格式如下:

```
<jsp:forward page="url"/>
```

(3) <jsp:useBean>：该操作用来在JSP页面中创建一个JavaBean实例，并指定它的名字和作用范围。其常用的语法格式如下：

```
<jsp:useBean id="id" scope="page|request|session|application"class="className"/>
```

其中，id用来表示实例的名字；scope表示此对象可以使用的范围；class指定对象的类型。

(4) <jsp:setProperty>：jsp:setProperty与useBean一起协作，用来设置JavaBean的属性。其常用的语法格式如下：

```
<jsp:setProperty name="beanName" property="propertyName" value="propertyValue"/>
```

其中，name用来指定要赋值的JavaBean的名字，property指定属性名，value指定属性值。

(5) <jsp:getProperty>：该操作用来访问一个JavaBean的属性值。它访问的属性值将转换成一个String，然后发送到输出流中。如果属性是一个对象，将调用toString()方法。其常用的语法格式如下：

```
<jsp:setProperty name="beanName" property="propertyName" />
```

12.2.3 JSP内建对象

JSP为了简化页面的开发提供了一些内部对象。这些内部对象不需要由JSP的编写者实例化，它们由容器实现和管理，在所有的JSP页面中都能使用内部对象。常见的JSP内部对象有out对象、request对象、response对象、session对象和application对象。

1. out对象

out对象主要用来向客户端输出数据。out对象表示为客户打开的输出流，可以使用out对象的print()、println()方法向客户端发送数据。

2. request对象

request对象代表请求对象，通过getParameter()方法可以得到request的参数。来自客户端的请求经Servlet容器处理后，由request对象进行封装。它作为jspService()方法的一个参数由容器传给JSP页面。request对象被包装成HttpServletRequest接口，相关方法介绍参见12.3.2节。

3. response对象

response对象封装了JSP产生的响应。和request对象一样，它由容器生成，作为jspService()方法的一个参数传入JSP页面。因为输出流是缓冲的，所以可以设置HTTP状态码和response头。response对象被包装成HttpServletResponse接口，相关方法介绍参见12.3.2节。

4. session对象

session对象用来保存每个用户信息，以便跟踪每个用户的操作状态。其中，session信息保存在服务器端，session的id保存在客户端的Cookie中。一般情况下，用户首次登录系统时容器会给此用户分配一个唯一标识session id，这个id用于区分其他用户，当用户退出系统时，这个session会自动消失。通过session.setAttribute()方法把相关信息保存在session中，通过session.getAttribute()方法从session中获取信息。

5. application 对象

application 对象为多个应用程序保存信息。对于一个容器而言,每个用户都共同使用一个 application 对象,这和 session 对象是不一样的。服务器启动后会自动创建 application 对象,这个对象会一直保持,直到服务器关闭为止。通过 application.setAttribute()方法把相关信息保存在 application 中,通过 application.getAttribute()方法从 application 中获取信息。

【例 12.4】 使用 request 对象的例子。

数据输入的静态网页(input.html)如下:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>数据输入</title>
</head>
<body>
<form method="post action"="requestDemo.jsp">
<table>
    <tr>
        <td>请输入登录名:</td>
        <td><input type="text" name="name"></td>
    </tr>
    <tr>
        <td>请输入密码:</td>
        <td><input type="password" name="password"></td>
    </tr>
    <tr>
        <td><input type="submit" value="登录"></td>
    </tr>
</table>
</form>
</body>
</html>
```

requestDemo.jsp 页面代码如下:

```
<%@ page language="java" contentType="text/html; charset=gb2312"
    pageEncoding="gb2312" import="java.io.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>request 对象的使用</title>
</head>
<body>
```

```
Request 对象的信息：
<hr>
<%
    out.println("<br> getMethod:");
    out.println(request.getMethod());
    out.println("<br> getParameter:");
    out.println(request.getParameter("name"));

    out.println("<br> getAttributeNames:");
    java.util.Enumeration e= request.getAttributeNames();
    while(e.hasMoreElements())
        out.println(e.nextElement());

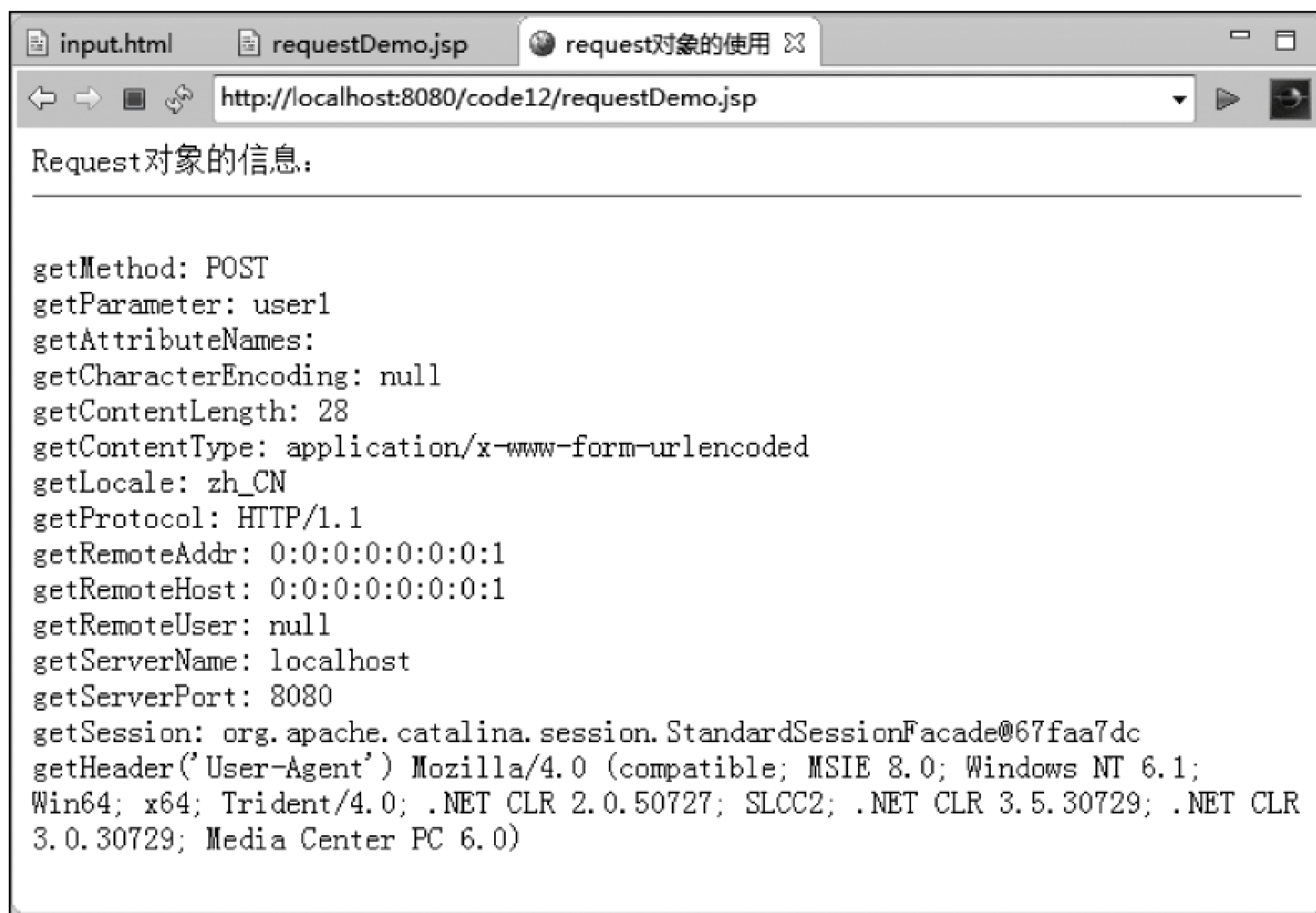
    out.println("<br> getCharacterEncoding:");
    out.println(request.getCharacterEncoding());
    out.println("<br> getContentLength: ");
    out.println(request.getContentLength());
    out.println("<br> getContentType:");
    out.println(request.getContentType());
    out.println("<br> getLocale:");
    out.println(request.getLocale());
    out.println("<br> getProtocol:");
    out.println(request.getProtocol());
    out.println("<br> getRemoteAddr:");
    out.println(request.getRemoteAddr());
    out.println("<br> getRemoteHost:");
    out.println(request.getRemoteHost());
    out.println("<br> getRemoteUser:");
    out.println(request.getRemoteUser());
    out.println("<br> getServerName:");
    out.println(request.getServerName());
    out.println("<br> getServerPort:");
    out.println(request.getServerPort());
    out.println("<br> getSession:");
    out.println(request.getSession(true));
    out.println("<br> getHeader('User-Agent')");
    out.println(request.getHeader("User-Agent"));
%>
</body>
</html>
```

程序可能的运行结果如图 12-4 所示。

分析：在 input.html 中通过表单将用户输入的数据发送给 requestDemo.jsp 页面，这些数据被封装在 request 对象中。调用 request 对象的相关方法可以获取相关数据和属性值。



(a)



(b)

图 12-4　例 12.4 程序可能的运行结果

【例 12.5】　在 response 中使用 Cookie 的例子。

```
<%@ page language="java" contentType="text/html; charset=gb2312"
    pageEncoding="gb2312" import="java.util.*"%>
<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>在 response 中使用 Cookie</title>
</head>
<body>
<%
    String userName="zhangshan";
    Cookie[] cookie=request.getCookies();
    Cookie cookie_response=null;
    List list=Arrays.asList(cookie);
    Iterator it=list.iterator();
    while(it.hasNext()) {
        Cookie temp=(Cookie) it.next();
        if (temp.getName().equals(userName+"_access_time")) {
```

```
                cookie_response= temp;
                break;
            }
        }

        out.println("当前的时间: "+ new Date()+ "<br> ");
        if (cookie_response != null) {
            out.println("上一次访问的时间: "+ cookie_response.getValue());
            cookie_response.setValue(new Date().toString());
        } else {
            cookie_response= new Cookie(userName+ "_access_time",
                    new Date().toString());
        }
        response.addCookie(cookie_response);
        response.setContentType("text/html");
        response.flushBuffer();
%>
</body>
</html>
```

程序可能的运行结果如图 12-5 所示。

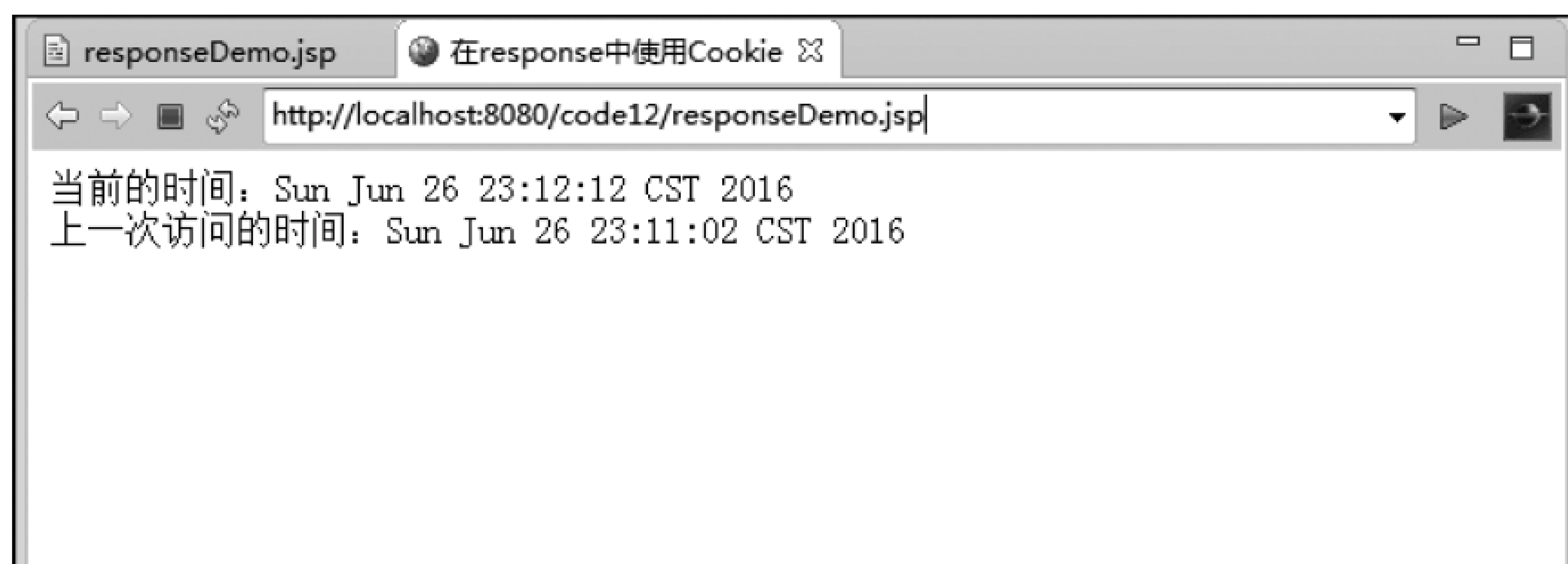


图 12-5　例 12.5 程序可能的运行结果

分析:在上述程序中,使用了一个 Cookie,该 Cookie 记录了用户上一次访问网页的时间。每次用户登录时,JSP 页面就通过 request 对象把客户端的所有 Cookie 获取过来,然后读取 Cookie 的值,如果有用户曾经访问过该网页,那么 request 对象中应该包含这个 Cookie,此时读取该 Cookie 的值,并且在网页中显示出来。最后更新 Cookie 的值,把它发送给客户端。

12.2.4　实用案例

【例 12.6】　商品信息展示。

JavaBean 代码如下:

```
package code1202;
public class CommodityInfoBean {
    private String name;
```

```
    private Float price;
    private String status;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name=name;
    }

    public Float getPrice() {
        return price;
    }

    public void setPrice(Float price) {
        this.price=price;
    }

    public String getStatus() {
        return status;
    }

    public void setStatus(String status) {
        this.status=status;
    }

    public CommodityInfoBean() {
        this.name="惠普笔记本电脑 CQ515";
        this.price=2950.0f;
        this.status="在售";
    }
}
```

showCommodityList.jsp 页面代码如下:

```
<%@page language="java" contentType="text/html; charset=gb2312"
    pageEncoding="gb2312"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>显示商品信息</title>
</head>
<body>
<jsp:useBean id="commodity" scope="page" class="code1202.CommodityInfoBean"/>
```

```
<h1>商品信息如下:</h1>
名称:<%=commodity.getName()%>
<br/>
价格:<%=commodity.getPrice()%>
<br/>
状态:<jsp:getProperty property="status" name="commodity"/>
<br/>
</body>
</html>
```

程序运行结果如图 12-6 所示。



图 12-6　例 12.6 程序运行结果

分析：商品信息的展示用一个 JSP 页面来实现，其中 CommodityInfoBean 类作为一个 JavaBean，存储了商品信息。通过 jsp：useBean 在页面中创建 CommodityInfoBean 类的一个实例，并通过两种方式读取 JavaBean 中的信息：一种方式是通过 JSP 表达式<%=commodity.getName()%>读取 JavaBean 中的信息；另一种方式是通过 jsp：getProperty 来读取 JavaBean 的属性。

12.3　Servlet 技术

12.3.1　Servlet 介绍

Servlet(Java 服务器小程序)是用 Java 编写的服务器端程序，是由服务器端调用和执行的、按照 Servlet 自身规范编写的 Java 类。Servlet 可以看成是用 Java 编写的 CGI，但是它的功能和性能比 CGI 更加强大。

Servlet 带给开发人员最大的好处是它可以处理客户端传来的 HTTP 请求，并返回一个响应。Servlet 是一个 Java 类，Java 语言能够实现的功能，Servlet 基本上都能实现(除了图形界面外)。

Servlet 部署在容器里，它的生命周期由容器管理。Servlet 的生命周期可以概括为以下几个阶段。

(1) 装载 Servlet：这项操作一般是动态执行的。有些服务器提供了相应的管理功能。可以在启动的时候就装载 Servlet 并能够初始化特定的 Servlet。

(2) 创建一个 Servlet 实例。

(3) 调用 Servlet 的 init()方法。

(4) 服务：如果容器接收到对此 Servlet 的请求，那么它调用 Servlet 的 service()方法。

(5) 销毁：实例被销毁，通过调用 Servlet 的 destory()方法来销毁 Servlet。

在上述几个阶段中，对外提供服务是最重要的阶段，service()方法是人们最关心的方法，因为它才是真正处理业务的方法。Servlet 为客户端提供服务的过程，如图 12-7 所示。

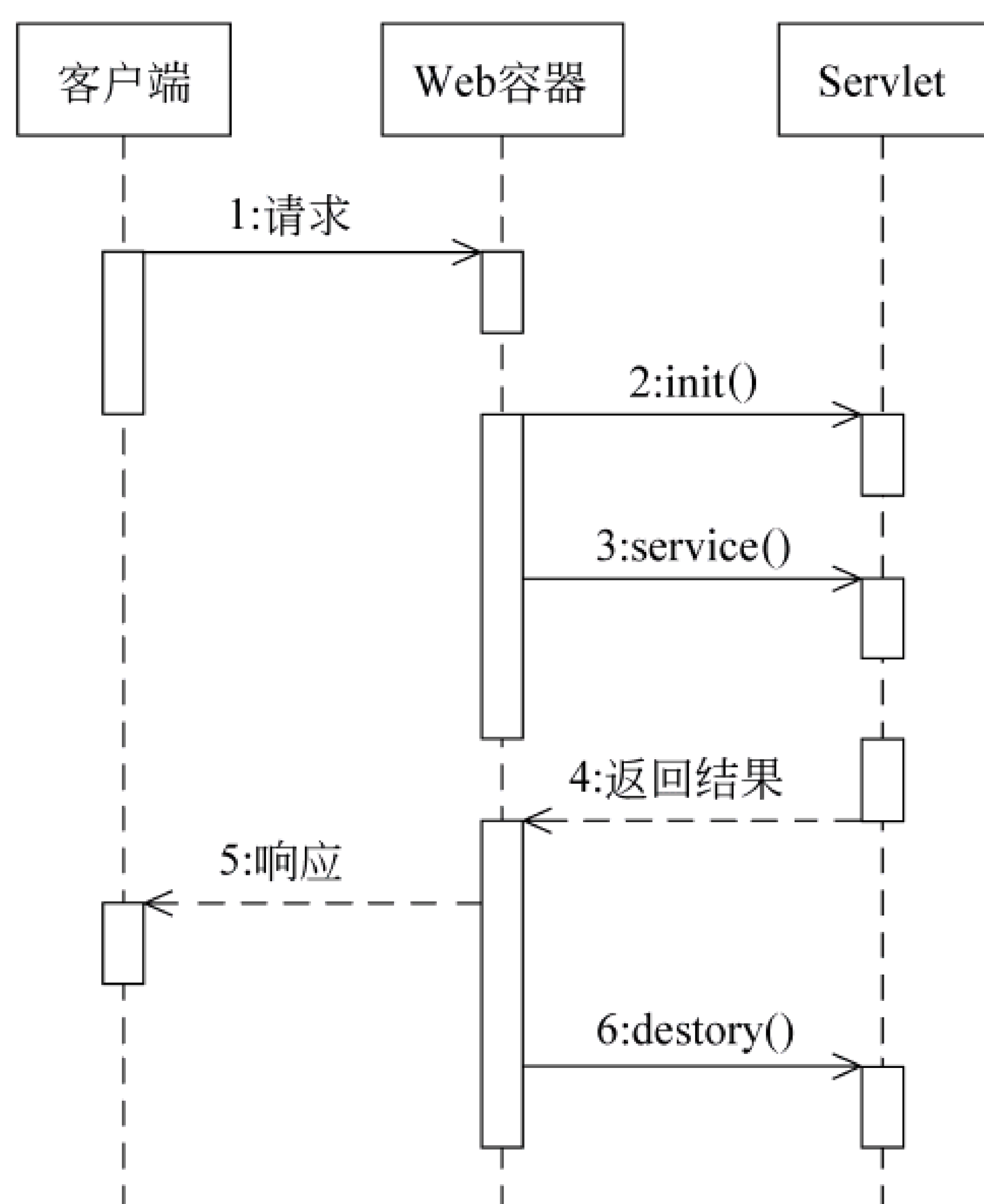


图 12-7　Servlet 对客户端提供服务的过程

将 Servlet 部署在容器中时，需要在部署描述符 web.xml 中进行 Servlet 配置。典型的 Servlet 配置包含 Servlet 的名字、Servlet 的类、初始化参数和 Servlet 的映射等，写法如下：

```
<servlet>
    <description>
    </description>
    <display-name>servletName</display-name>
    <servlet-name>servletName</servlet-name>
    <servlet-class>
    packageName.servletClassName</servlet-class>
</servlet>
```

在配置 Servlet 时，首先必须指定 Servlet 的名字和 Servlet 的类。此外还可以在<description>标记中添加关于 Servlet 的描述信息。

可以给一个 Servlet 做多个映射，这样就可以通过不同的方式来访问这个 Servlet。例如，可以对名为 servletName 的 Servlet 的映射配置如下：

```
<servlet-mapping>
    <servlet-name>servletName</servlet-name>
    <url-pattern>/servletName</url-pattern>
```

```
</servlet-mapping>
<servlet-mapping>
    <servlet-name>servletName</servlet-name>
    <url-pattern>/servletName/*</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>servletName</servlet-name>
    <url-pattern>/servletName.html</url-pattern>
</servlet-mapping>
```

通过这些配置,可以使用不同的方式来访问这个 Servlet。有趣的是对于第二种映射方式,以下的访问都是有效的:

```
http://localhost:8080/ch12/servletName/afasfas?dafd
http://localhost:8080/ch12/servletName/servletName
http://localhost:8080/ch12/servletName/servletName.jsp
http://localhost:8080/ch12/servletName/servletName.jsp?name=abc
```

也就是说,只要以/servletName/开头,都能访问到这个 Servlet。

12.3.2 Servlet 常用接口的使用

Servlet 的类接口可以从以下几个方面进行分类。

(1) Servlet 实现相关:定义了用于实现 Servlet 相关的类和方法。

(2) Servlet 配置相关:主要包括 ServletConfig 接口。

(3) Servlet 异常相关:Servlet API 定义了两个异常,分别是 ServletException 和 UnavailableException。

(4) 请求和相应相关:用于接收客户端的请求,并做出相应的响应。

(5) 会话跟踪:用于跟踪与客户端的会话。

(6) Servlet 上下文:通过这个接口,可以在多个 Web 应用程序中共享数据。

(7) Servlet 协作:主要是 RequestDispatcher 接口,用于进行视图派发。

1. Servlet 实现相关

下面介绍和 Servlet 实现相关的类和接口。

1) Servlet

声明:public interface Servlet

该接口是所有 Servlet 必须直接或者间接实现的接口。它定义了以下方法。

void init(ServletConfig config):初始化 Servlet。

void destroy():销毁 Servlet。

String getServletInfo():获得 Servlet 信息。

ServletConfig getServletConfig():获得 Servlet 配置相关信息。

void service(ServletRequest req, ServletResponse res):运行应用程序逻辑的入口点,它接收两个参数,ServletRequest 表示客户端请求的信息,ServletResponse 表示对客户端的响应。

2）GenericServlet

声明：public abstract class GenericServlet extends Object implements Servlet，ServletConfig，java. io. Serializable

该抽象类提供了对 Servlet 接口的基本实现。它的 service()方法是一个抽象方法，GenericServlet 的派生类必须直接或者间接实现这个方法。

3）HttpServlet

声明：public abstract class HttpServlet extends GenericServlet implements java. io. Serializable

HttpServlet 类是针对使用 HTTP 协议的 Web 服务器的 Servlet 类，能够提供 HTTP 协议的功能。

HttpServlet 的子类必须实现以下方法中的一个。

void doGet(HttpServletRequest req，HttpServletResponse resp)：支持 HTTP Get 请求。

void doPost(HttpServletRequest req，HttpServletResponse resp)：支持 HTTP Post 请求。

void doPut(HttpServletRequest req，HttpServletResponse resp)：支持 HTTP Put 请求。

void doDelete(HttpServletRequest req，HttpServletResponse resp)：支持 HTTP Delete 请求。

2. Servlet 配置相关

javax. servlet. ServletConfig 接口代表了 Servlet 的配置。

声明：public interface ServletConfig

这个接口的主要方法有以下几个。

String getInitParameter(java. lang. String name)：返回特定名字的初始化参数。

java. util. Enumeration getInitParameterNames()：返回所有的初始化参数的名字。

ServletContext getServletContext()：返回 Servlet 的上下文对象的引用。

3. Servlet 异常相关

1）ServletException

声明：public class UnavailableExceptionextends ServletException

它包含几个构造方法和一个获得异常原因的方法，这个方法如下。

java. lang. Throwable getRootCause()：返回造成这个 ServletException 的原因。

2）UnavailableException

声明：public class UnavailableExceptionextends ServletException

当 Servlet 暂时或者永久不能使用时，就会抛出这个异常。

4. 请求和相应相关

和请求响应相关的类和接口非常多，这里重点介绍 HttpServletRequest 和 HttpServletResponse 两个接口。

1）HttpServletRequest

声明：public interface HttpServletRequest extends ServletRequest

这个接口中最常用的方法就是获取请求中的参数，这个请求中的参数是客户端表单中的数据。HttpServletRequest 接口可以获取由客户端传送的参数名称，也可以获取客户端正在使用的通信协议，可以获取产生请求并且接收请求的服务器远端主机名及其 IP 地址等一些这样的信息。

JSP 中的内建对象 request 实质上是一个 HttpServletRequest 实例。下面是 HttpServletRequest 接口的一些重要方法。

Cookie[] getCookies()：获得客户端发送的 Cookie。

HttpSession getSession()：返回和客户端关联的 Session，如果没有客户端分配 Session，则返回 null。

String getParameter(String name)：获得请求中名为 name 的参数的值，如果请求中没有这个参数，则返回 null。

String[] getParameterValues(String name)：返回请求中名为 name 的参数值，这个参数值往往是 checkbox 或者 select 控件提交的，获得的值是一个 String 数组。

Object getAttribute(String name)：返回由 name 指定的属性值，如果指定的属性值不存在，则会返回一个 null 值。

java.util.Enumeration getAttributeNames()：返回 request 对象的所有属性的名字集合，其结果是一个枚举的实例。

String getCharacterEncoding()：返回请求中的字符编码方式。

int getContentLength()：返回请求的 Body 的长度，如果不确定长度，则返回－1。

String getContentType()：返回请求的 MIME 类型，如果类型不确定，则返回－1。

java.util.Locale getLocale()：返回客户端所在的地域信息。

String getProtocol()：获取客户端向服务器端传送数据所依据的协议名称。

String getRemoteAddr()：获取客户端的 IP 地址。

String getRemoteHost()：获取客户端的名字。

int getRemotePort()：获取客户端的端口号。

String getServerName()：获取服务器的名字。

int getServerPort()：获取服务器的端口号。

void removeAttribute(String name)：删除请求中的一个属性。

void setAttribute(String name，Object o)：设置名字为 name 的 request 参数值，该值由 Object 类型的参数 o 指定。

2）HttpServletResponse

声明：public interface HttpServletResponse extends ServletResponse

它代表了对客户端的 HTTP 响应。HttpServletResponse 接口允许 Servlet 设置内容长度和响应的 MIME 类型，并且提供输出流 ServletOutputStream。常用的方法有以下几种。

void addCookie(Cookie cookie)：在响应中增加一个 Cookie。

String encodeURL(String url)：使用 URL 和一个 sessionId 重写这个 URL。

void sendRedirect(String location)：把响应发送到另一页面或者 Servlet 进行处理。

void setContentType(String type)：设置响应的 MIME 类型。

void setCharacterEncoding(String charset):设置响应的字符编码类型。

void flushBuffer():强制把当前缓冲区的内容发送到客户端。

ServletOutputStream getOutputStream():返回到客户端的输出流对象。

void sendError(int sc):向客户端发送错误的信息。例如,404 是指网页不存在或者请求的页面无效。

void setHeader(String name,String value):设置指定名字的 Http 文件头的值,如果该值已经存在,则新值会覆盖原有的旧值。

5. 会话跟踪

和会话跟踪相关的类和接口是 HttpSession。

声明:public interface HttpSession

这个接口被 Servlet 引擎用来实现 Http 客户端和 Http 会话两者之间的关联。这种关联可能在多处连续和请求中持续一段给定的时间。Session 用来在无状态的 Http 协议下越过多个请求页面来维持状态和识别用户。

一个 Session 可以通过 Cookie 或者重写 URL 来维持,常用方法有以下几种。

long getCreationTime():返回创建 Session 的时间。

String getId():返回分配给这个 Session 的标识符。一个 Http Session 的标识符是一个由服务器来建立和维护的唯一的字符串。

long getLastAccessedTime():返回客户端最后一次发出与这个 Session 有关的请求的时间。如果这个 Session 是新建立的,则返回−1。

int getMaxInactiveInterval():返回一个秒数,这个秒数表示客户端在不发出请求时,Session 被 Servlet 引擎维持的最长时间。在这个时间之后,Session 可能被 Servlet 引擎终止。如果这个 Session 不会被终止,则返回−1。

Object getAttribute(String name):返回一个以给定的名字绑定到 Session 上的对象。如果不存在这样的绑定,则返回空值。

java.util.Enumeration getAttributeNames():返回绑定到 Session 上的所有对象的名称集合。

void invalidate():终止 Session,所有绑定到该 Session 上的数据都会被清除。

boolean isNew():判断 Session 是不是新的。如果一个 Session 已经被服务器建立,但还没有收到相应客户端的请求,则这个 Session 被认为是新的。

void setAttribute(String name, Object value):以给定的名字绑定给定的对象到 Session 中。已存在的同名数据会被新的数据覆盖。

void removeAttribute(String name):取消给定名字的对象在 Session 上的绑定。如果未找到给定名字的绑定对象,则该方法什么都不做。

void setMaxInactiveInterval(int interval):设置一个秒数,这个秒数表示客户端在不发出请求时,Session 被 Servlet 引擎维持的最长时间。

6. Servlet 上下文

和 Servlet 上下文相关的接口有 ServletContext。

声明:public interface ServletContext

ServletContext 对象表示一组 Servlet 共享的资源。常用的方法如下。

Object getAttribute(String name)：获得 ServletContext 中名称为 name 的属性。

ServletContext getContext(String uripath)：返回给定的 uripath 的应用的 Servlet 上下文。

void removeAttribute(String name)：删除名称为 name 的属性。

void setAttribute(String name,Object object)：在 ServletContext 中设置一个属性，这个属性的名称为 name，值为 object 对象。

7．Servlet 协作

Servlet 协作主要是 RequestDispatcher 接口，它可以把一个请求转发给另一个 Servlet。

声明：public interface RequestDispatcher

它包含的方法如下：

void forward(ServletRequest request, ServletResponse response)：把请求转发给服务器上的另一个资源(Servlet、JSP 或 HTML)。

void include(ServletRequest request, ServletResponse response)：把服务器上的另一个资源(Servlet、JSP 或 HTML)包含到响应中。

12.3.3 使用 HttpServlet 处理客户端请求

Servlet 被设计成是由请求驱动的。Servlet 的请求可能包含多个数据，当 Web 容器接收到某个对 Servlet 的请求时，把它封装成一个 HttpServletRequest 对象，然后把此对象传给 Servlet 对应的服务方法，服务方法通常是 doGet 和 doPost 方法。

1．doGet

Get 调用用于获取服务器信息，并将其作为响应返回客户端。当经由 Web 浏览器或者通过 HTML、JSP 直接访问 Servlet 的 URL 时，一般使用 Get 调用。Get 调用在 URL 里显示正在传送给 Servlet 的数据。

【例 12.7】 通过 Get 调用实现 Servlet 与网页中表单的交互。

网页代码如下：

```
<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>使用 Get 调用来传递参数</title>
</head>
<body>
<form action="doGetDemo" method="get">
请输入参数：
<input type="text" name="name" />
<input type="submit" value="提交" />
</form>
</body>
</html>
```

Servlet 代码如下：

```
package code1203;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name="doGetDemo", urlPatterns="/doGetDemo")
public class DoGetDemo extends HttpServlet {
    private static final long serialVersionUID=1L;

    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
        request.setCharacterEncoding("gb2312");
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("获得了一个参数值:name=<br>"+request.getParameter
        ("name"));
        out.flush();
    }
}
```

程序运行结果如图 12-8 所示。



(a)



(b)

图 12-8 例 12.7 程序运行结果

分析：通过 Get 调用传递参数时，在客户端的 form 中必须指定调用的类型是 Get。通过 Servlet 的 doGet()方法来处理请求，并通过 request.getParameter()方法来获得请求中的参数。

注意：在程序的运行结果中，请求的参数自动添加到了浏览器的地址栏中，这样可能会带来安全性方面的问题。

2. doPost

doPost 也用于客户端把数据传送给服务器端。使用它的好处是可以隐藏发送给服务器端的任何数据。Post 适合于发送大量的数据。

【例 12.8】 通过 Post 调用实现 Servlet 与网页中表单的交互。

网页代码如下：

```
<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>使用 Post 调用来传递参数</title>
</head>
<body>
<form action="doPostDemo" method="post">
请输入参数:<br/>
<textarea rows="10" cols="50" name="data"></textarea><br/>
<input type="submit" value="提交" />
</form>
</body>
</html>
```

Servlet 代码如下：

```
package code1203;

import java.io.IOException;
import java.io.PrintWriter;

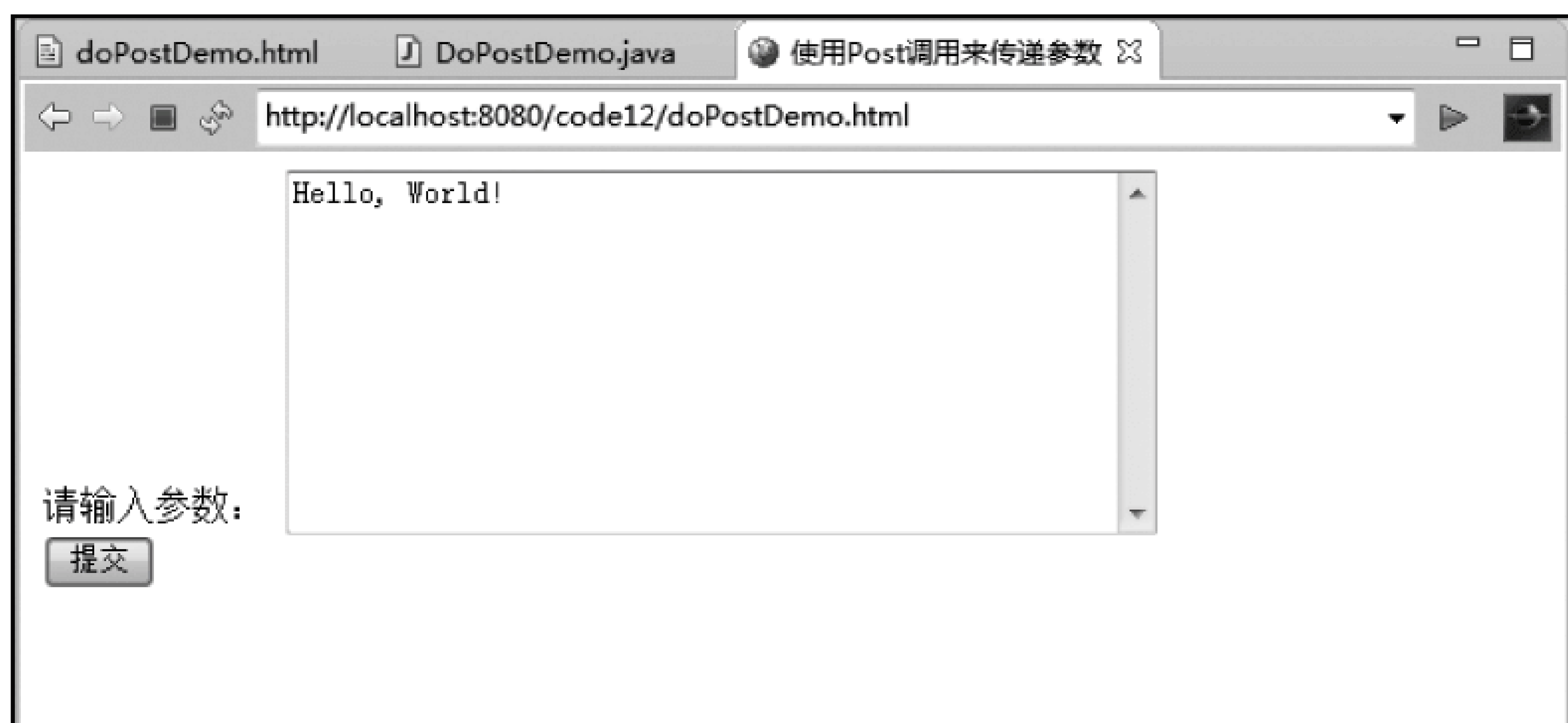
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/doPostDemo")
public class doPostDemo extends HttpServlet {
    private static final long serialVersionUID=1L;

    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
```

```
            request.setCharacterEncoding("gb2312");
            response.setContentType("text/html;charset=gb2312");
            PrintWriter out=response.getWriter();
            out.println("获得了一个参数值:data=<br>"+request.getParameter
            ("data"));
            out.flush();
        }
}
```

程序运行结果如图 12-9 所示。



(a)



(b)

图 12-9 例 12.8 程序运行结果

分析:通过 Post 调用传递参数时,在客户端的 form 中必须指定调用的类型是 Post。通过 Servlet 的 doPost()方法来处理请求,并通过 request.getParameter()方法来获得请求中的参数。可以看到这里提交的数据不会在浏览器的地址栏中显示。

12.3.4 获得 Servlet 初始化参数

Servlet 可以配置一些初始化参数,可以在 Servlet 中获得这些初始的参数。下面来看一个具体的例子。

【例 12.9】 假如 Servlet 需要知道应用程序的名字,可以把应用程序名 appName 配置为初始化参数(initParams),然后在 Servlet 执行时来获取初始化参数 appName 的值。

Servlet 代码如下:

```
package ch1203;
```

```
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(
        urlPatterns={ "/initParameterDemo" },
        initParams={
                @WebInitParam(name="appName", value="第 12 章示例程序")
        })
public class InitParameterDemo extends HttpServlet {
    private static final long serialVersionUID=1L;
    private String appName;

    public void init() throws ServletException {
        appName=getInitParameter("appName");
    }

    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        response.setCharacterEncoding("gb2312");
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("应用程序名是：");
        out.println(appName);
    }

    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }

}
```

程序运行结果如图 12-10 所示。

分析：在 Servlet 类上使用@WebServlet 注解，对其<init-param>属性设置了 Servlet 的初始化参数，而在 Servlet 中通过 getInitParameter()方法来获取初始化参数值。

12.3.5 实用案例

【例 12.10】 简单的用户问好功能。

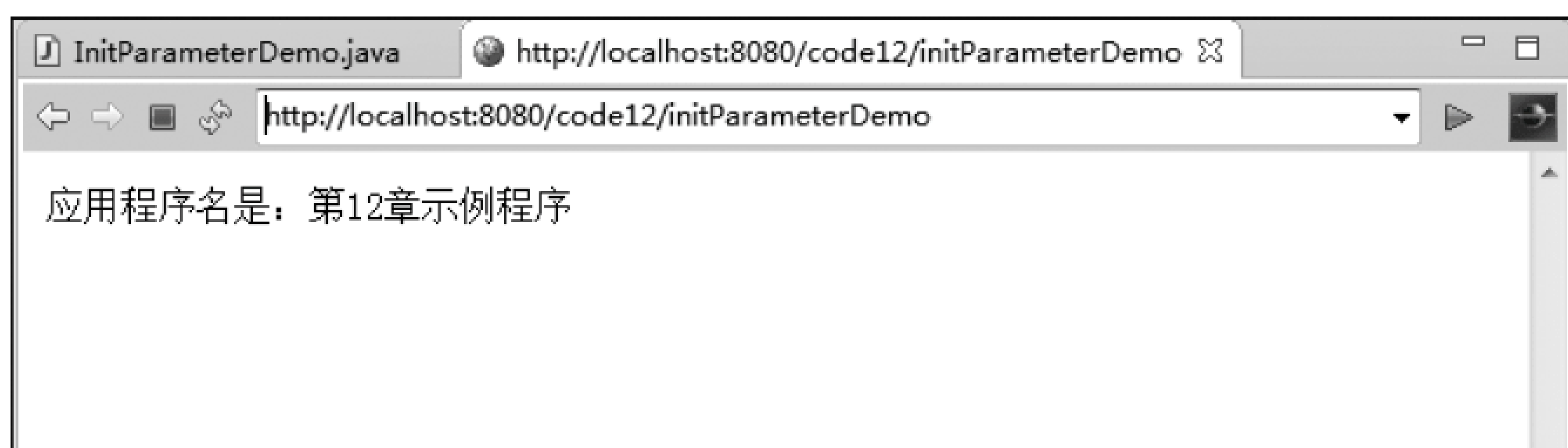


图 12-10　例 12.9 程序运行结果

Servlet 代码如下：

```
//GreetingDemo.java
package ch1203;

import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/greetingDemo")
public class GreetingDemo extends HttpServlet {
    private static final long serialVersionUID=1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        doPost(request,response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        String userName="Jack";
        request.getSession().setAttribute("userName", userName);
        RequestDispatcher rd=getServletContext().getRequestDispatcher
        ("/showGreetingInfo");
        rd.forward(request, response);
    }

}

//ShowGreetingInfo.java
package ch1203;
```

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/showGreetingInfo")
public class ShowGreetingInfo extends HttpServlet {
    private static final long serialVersionUID=1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        doPost(request, response);
    }

      protected void doPost (HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        String userName= (String)request.getSession().getAttribute
        ("userName");
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("你好,"+userName);
        out.flush();
    }
}
```

程序运行结果如图12-11所示。

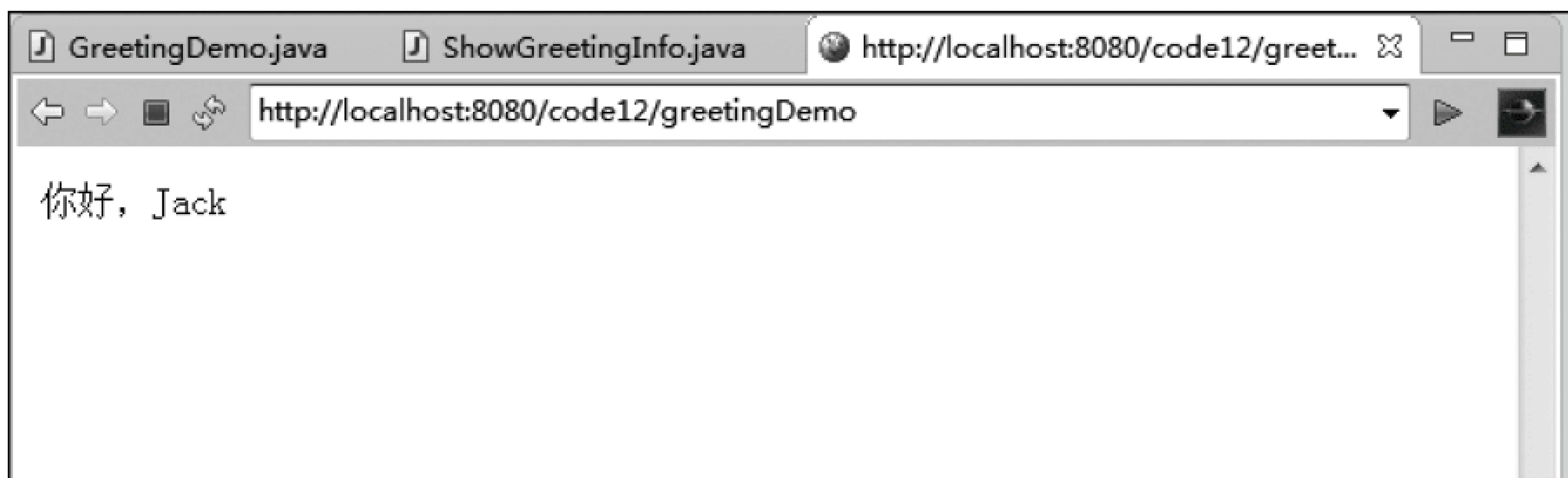


图12-11　例12.10程序运行结果

分析：上例中请求的处理分成了两步来完成。第一步由GreetingDemo.java来实现，主要是将用户名Jack存入session中；第二步由ShowGreetingInfo.java来实现，它从session中取出用户名变量，并将最后的显示结果传回客户端。这里两个Servlet之间的相互协作是通过RequestDispatcher和session来完成的，其中RequestDispatcher实现Servlet

之间控制流的跳转,session 作为传递数据的存储空间。

12.4 JSP 和 Servlet 结合的方法

JSP 网站开发技术标准给出了两种使用 JSP 的方法:模式一和模式二。模式一是 JSP＋JavaBean 的结合,模式二是 JSP＋JavaBean＋Servlet 的结合。在当今的软件应用开发中,比较偏向于使用模式二,但是模式一在小型应用开发中比较占优势。JSP 模式一和模式二如图 12-12 所示,图 12-12(a)是模式一,图 12-12(b)是模式二。下面来介绍这两种模式。

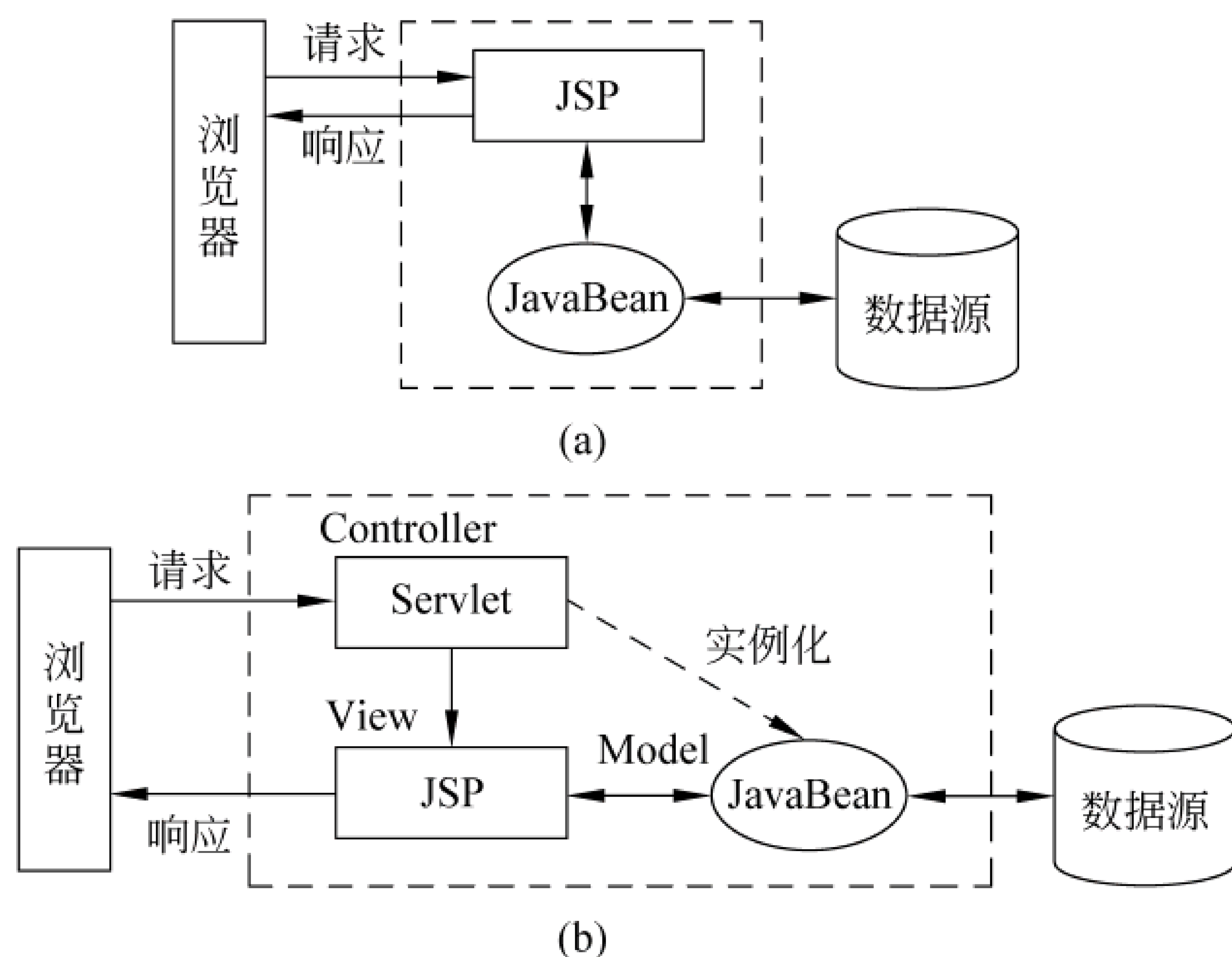


图 12-12 JSP 模式一和模式二

12.4.1 模式一:JSP＋JavaBean

在模式一中,JSP 页面独自响应请求并将处理结果返回客户。所有的数据通过 JavaBean 来处理,JSP 实现页面的展现。模式一技术实现了页面展现与页面商业逻辑的分离。但是大量使用此模式时,可能带来一个副作用会导致在页面里嵌入大量的 Java 控制代码,当要处理的业务逻辑复杂时这种情况会变得非常糟糕。所以在大型的项目里,这种方式将会导致页面维护困难。一般在小型的应用中可以考虑采用此模式。

12.4.2 模式二:JSP＋Servlet＋JavaBean

在模式二中,结合了 JSP 和 Servlet 技术。模式二充分利用了 JSP 和 Servlet 两种技术的优点。此模式遵循视图控制器模式(Model View Controler,MVC),它的主要思想是使用一个或多个 Servlet 作为控制器。请求由前台的 Servlet 接收并处理后,会重新定向到 JSP。在 Servlet 作为控制器时,每个 Servlet 通常只实现很少一部分功能,多个 Servlet 控制器就可以结合起来完成复杂的任务,这样的好处是 Servlet 的重用性好。而 JavaBean 作为模型的角色,它充当 JSP 和 Servlet 通信的中间工具,Servlet 处理完后设置 JavaBean 的属性,JSP 读取此 JavaBean 的属性,然后进行显示。在实际的项目开发中,页面设计者可以方便地使用 HTML 工具来开发 JSP 页面,而程序开发人员则可以用 Java 集成开发环境来开发

Servlet。模式二更加明显地将显示与逻辑处理分离开,适合于大型项目的开发。

12.4.3 JSP 和 Servlet 的选择

所有的 JSP 都必须编译成 Servlet,然后在 Servlet 容器中执行。从技术角度来看,JSP 和 Servlet 是一样的。

JSP 在以下几个方面要胜过 Servlet:

(1) JSP 以显示为中心,它为 Web 显示开发人员提供了更加方便的开发模式。

(2) JSP 借助 JavaBean,可以把显示与内容分离。

(3) JSP 由容器自动编译。

而 Servlet 则在以下方面发挥作用:

(1) 协调输出。

(2) 处理非常简单的业务逻辑。

(3) 处理 JSP 不好处理的后台服务或者其他有特殊要求的问题。

在特定的软件系统环境中,选择使用 JSP 还是 Servlet 通常不是绝对的。最常见的情况是把两者结合起来使用。例如在 JSP 模式二中,把 Servlet 作为视图控制器,让它来处理请求,当 Servlet 处理完请求后,把处理结果转发给 JSP,由 JSP 处理显示的问题。

12.4.4 实用案例

【例 12.11】 网站计数器功能。

Servlet 代码如下:

```
package code1204;

import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/websiteCounter")
public class WebsiteCounter extends HttpServlet {
    private static final long serialVersionUID=1L;
    private static int count;

    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        count++;
        //将访问次数写入 request 中,以便传给 JSP 页面
        request.setAttribute("hitCount", count);
        String url="/websiteCounterInfo.jsp";
```

```
            RequestDispatcher rd= getServletContext().getRequestDispatcher(url);
            rd.forward(request, response);
        }
    }
```

JSP 页面代码如下:

```
<%@ page language="java" contentType="text/html; charset=gb2312"
    pageEncoding="gb2312"%>
<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>网站计数器</title>
</head>
<body>
欢迎你,你是第
<%=request.getAttribute("hitCount")%>
位访问本网站的用户!
</body>
</html>
```

程序运行结果如图 12-13 所示。



图 12-13　例 12.11 程序运行结果

分析:通过一个静态变量 count 来记录网站的访问次数,每当网站被访问一次时,count 自增 1。在程序中,利用 Servlet 来处理用户的请求,并管理 count 的值;而利用 JSP 页面来显示相关信息。通过使用 RequestDispatcher 对象,实现从 Servlet 到 JSP 页面的跳转,把相应的 count 值作为 request 对象中的参数传递给 JSP 页面,以便在 JSP 页面中进行显示。

12.5　JSP 与 Servlet 开发实训任务

【任务描述】

编写一个简单的 Web 应用,实现用户登录的功能。

【任务分析】

用户登录主要包括两个操作步骤:①让用户在网页中输入用户名和密码;②对用户输

入的用户名和密码进行验证，判断是否是合法用户，最后将登录成功与否的结果返回给用户。这里采用 JSP 模式二的方法来实现用户登录功能，即编写一个 JavaBean（UserBean 类）作为模型，其中封装了验证合法用户的代码；编写一个 Servlet（UserLoginServlet 类）作为控制器，处理用户的请求；编写一个 JSP 页面（login. jsp）作为视图，显示用户的输入界面和登录结果界面。假定合法用户的用户名是 admin，密码是 admin。

【任务解决】

UserBean 类代码如下：

```
package code1205;

public class UserBean {
    private String validUserName;
    private String validUserPassword;

    public String getValidUserName() {
        return validUserName;
    }

    public void setValidUserName(String validUserName) {
        this.validUserName= validUserName;
    }

    public String getValidUserPassword() {
        return validUserPassword;
    }

    public void setValidUserPassword(String validUserPassword) {
        this.validUserPassword= validUserPassword;
    }

    public boolean isValidUser(String name, String password) {
        boolean result= false;
        if(validUserName.equals(name) && validUserPassword.equals(password)) {
            result= true;
        }
        return result;
    }
}
```

Servlet 类代码如下：

```
package code1205;

import java.io.IOException;
import java.io.PrintWriter;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/userLogin")
public class UserLogin extends HttpServlet {
    private static final long serialVersionUID=1L;

    protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        String name=request.getParameter("name");
        String password=request.getParameter("password");

        UserBean user=new UserBean();
        user.setValidUserName("admin");          //设置合法用户的信息
        user.setValidUserPassword("admin");

        response.setCharacterEncoding("gb2312");
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        if(user.isValidUser(name, password)){    //调用 JavaBean 的方法来验证用户
            out.println("登录成功!");
        }else{
            out.println("用户名或者密码错误,<a href='login.jsp'>请重试。</a>");
        }
    }

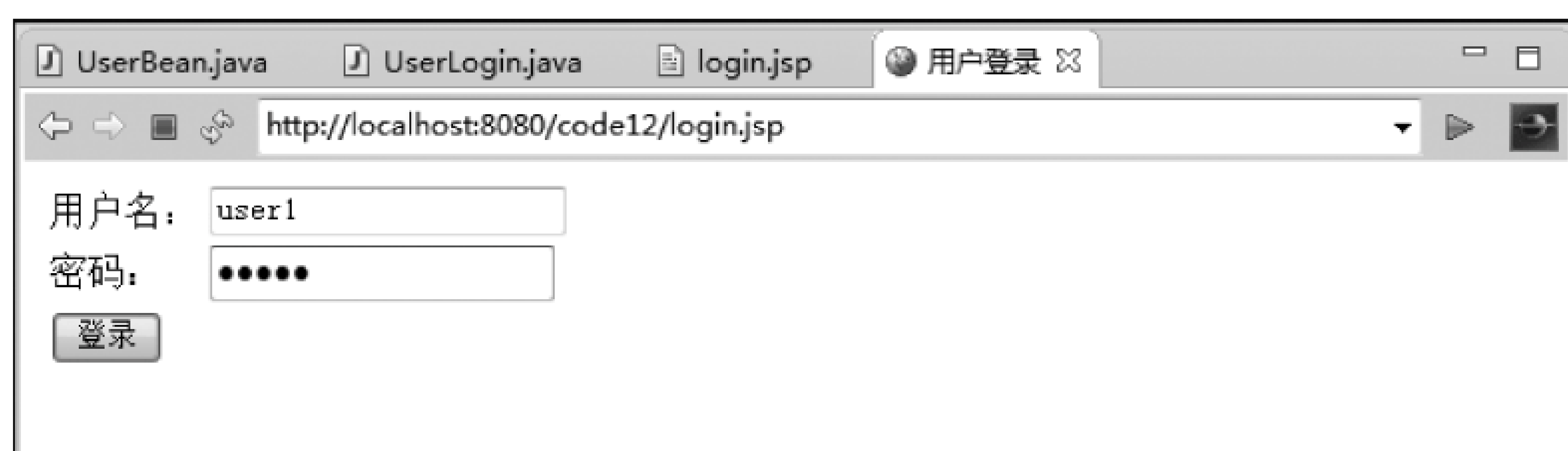
    protected void doPost(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {
        doGet(request,response);
    }
}
```

JSP 页面代码如下:

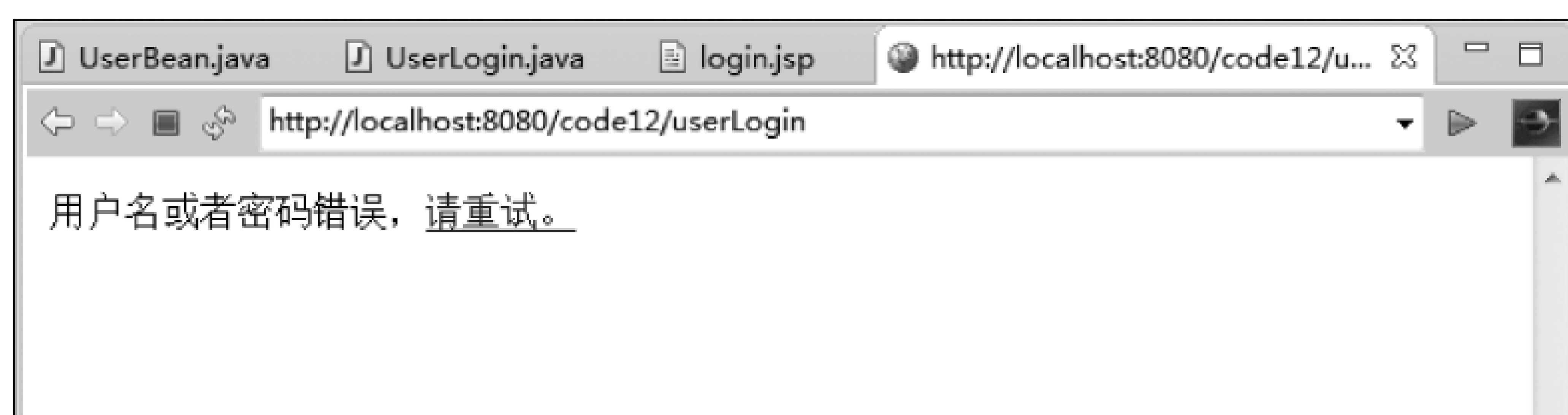
```
<%@ page language="java" contentType="text/html; charset=gb2312"
    pageEncoding="gb2312"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>用户登录</title>
</head>
```

```
<body>
<form action="userLogin" method="post">
<table>
    <tr>
        <td>用户名:</td>
        <td><input type="text" name="name" /></td>
    </tr>
    <tr>
        <td>密码:</td>
        <td><input type="password" name="password" /></td>
    </tr>
    <tr>
        <td colspan="2"><input type="submit" value="登录" /></td>
    </tr>
</table>
</form>
</body>
</html>
```

程序运行结果如图12-14所示。



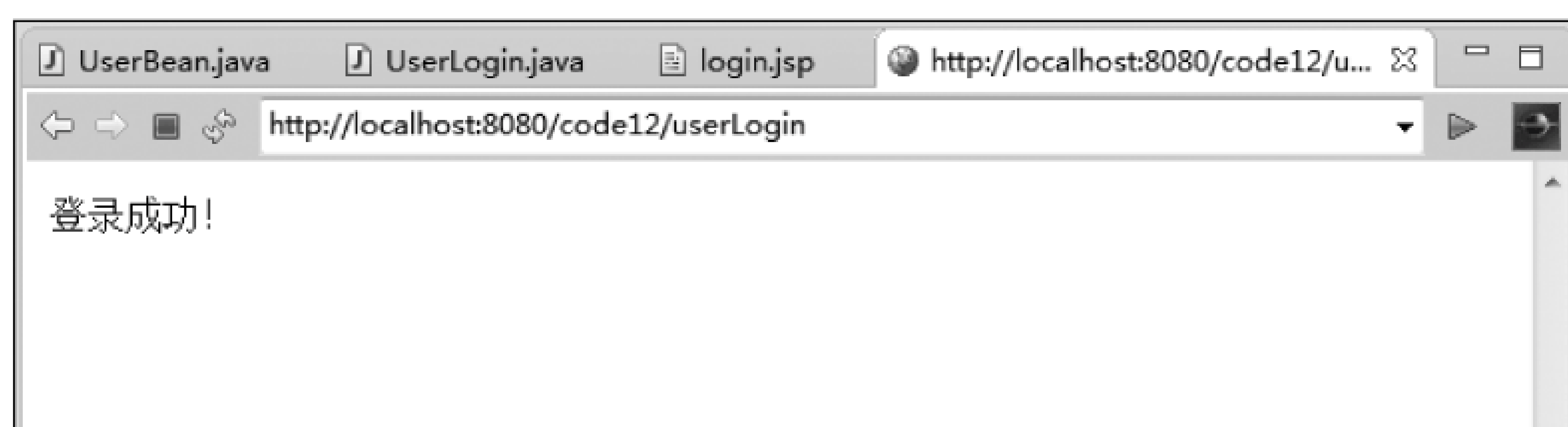
(a)



(b)



(c)



(d)

图 12-14 JSP 与 Servlet 开发实训任务运行结果

习题与思考

1. JSP 的工作原理是什么?
2. JSP 由哪些元素构成?
3. JSP 有哪些内置对象?
4. 如何将 JSP 与 Servlet 结合起来进行 Web 应用的开发?

第13章 用 Tomcat 构建 Web 站点

本章学习目标

Web 程序开发是 Java 最擅长的一个领域，也是目前工具最多、技术最复杂的一个领域。开发 Web 程序，首先要掌握如何使用 Web 服务器。所谓 Web 服务器，就是一种能够运行 Web 程序的软件系统。本章将以 Java 领域最流行的 Web 服务器为基础，结合 Eclipse IDE for Java EE Developers 讲解如下几个方面的知识：

（1）Tomcat 和 Eclipse IDE for Java EE Developers 的安装使用。

（2）如何在 Eclipse IDE for Java EE Developers 中结合 Tomcat 编写 Web 站点。

（3）如何在 Eclipse IDE for Java EE Developers 中用 Tomcat 运行 Web 站点。

（4）如何将 Web 站点发布到 Tomcat。

13.1 Tomcat 简介

Tomcat 是 Apache 软件基金会（Apache Software Foundation，http://www.apache.org）的 Jakarta 项目中的一个核心项目，是目前 Java 领域应用最广泛的轻量级 Web 应用服务器。在中小型系统和并发访问用户数量不大的场合普遍使用，并且是学习、开发和调试 Web 程序的首选。Tomcat 的主要功能是一个 JSP 和 Servlet 的容器。JSP 和 Servlet 是 Java 最主要的 Web 开发技术，详见第 12 章。

最初 Tomcat 由 Sun 公司的软件构架师詹姆斯·邓肯·戴维森开发。早期是为了给 Sun 公司正在开发的 JSP 和 Servlet 技术提供一个运行环境。后来他将其变为开源项目，并由 Sun 公司贡献给 Apache 软件基金会。由于 O'Reilly（国外著名的计算机类图书出版商）会为大部分开源项目出版一本相关的书，并且将其封面设计成某个动物的素描，因此当 O'Reilly 为 Tomcat 出书时，Tomcat 的作者希望将此项目以一个动物的名字命名。他希望这种动物能够自己照顾自己，最终他将其命名为 Tomcat。而 Tomcat 的 Logo 兼吉祥物也被设计成了一只公猫，如图 13-1 所示。Tomcat 一词在英文中的含义为“公猫”，它和动画片《猫和老鼠》并没有什么渊源。目前，Tomcat 最新的稳定版本为 9.0.0。

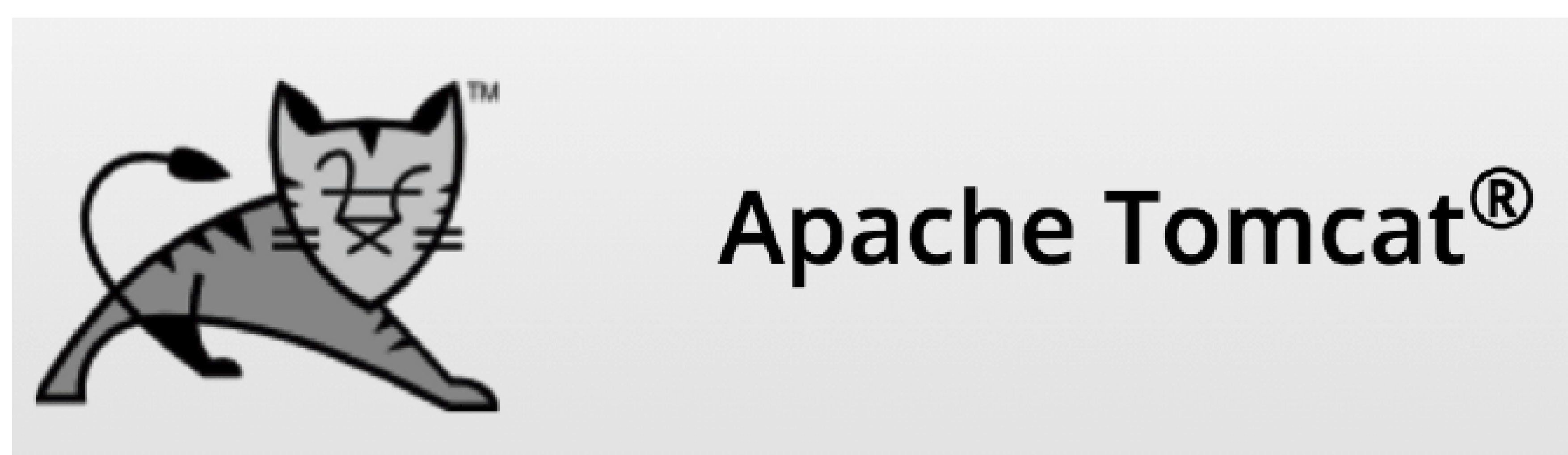


图 13-1　Apache Tomcat 的 Logo

13.2　安装配置 Tomcat

由于 Tomcat 是基于 Java 运行的程序,因此在安装 Tomcat 之前需要首先安装好 Java SE 环境,设置好 JAVA_HOME 和 PATH 环境变量。Tomcat 软件可以在 Tomcat 官方网站(http://tomcat.apache.org)下载,如图 13-2 所示,网站提供了二进制版本(Binary Distributions)和源代码版本(Source Code Distributions)供用户下载。通常如果只是使用 Tomcat 应该下载二进制版本,而如果想为 Tomcat 这个开源项目贡献力量或者想了解 Tomcat 的工作原理就可以下载源代码版本。二进制版本又分为 Core 和 Deployer 两个部分。Core 是 Tomcat 程序的核心部分,要使用 Tomcat 就必须下载。而 Deployer 提供了部署发布 Web 程序的辅助功能,属于选择下载的部分。Core 之下又分为多种类型的下载格式,本书选择下载 9.0.0 版本的 zip 格式。程序下载之后,将文件解压到任意目录即可完成安装(本书选择解压到 E:\开发工具)。

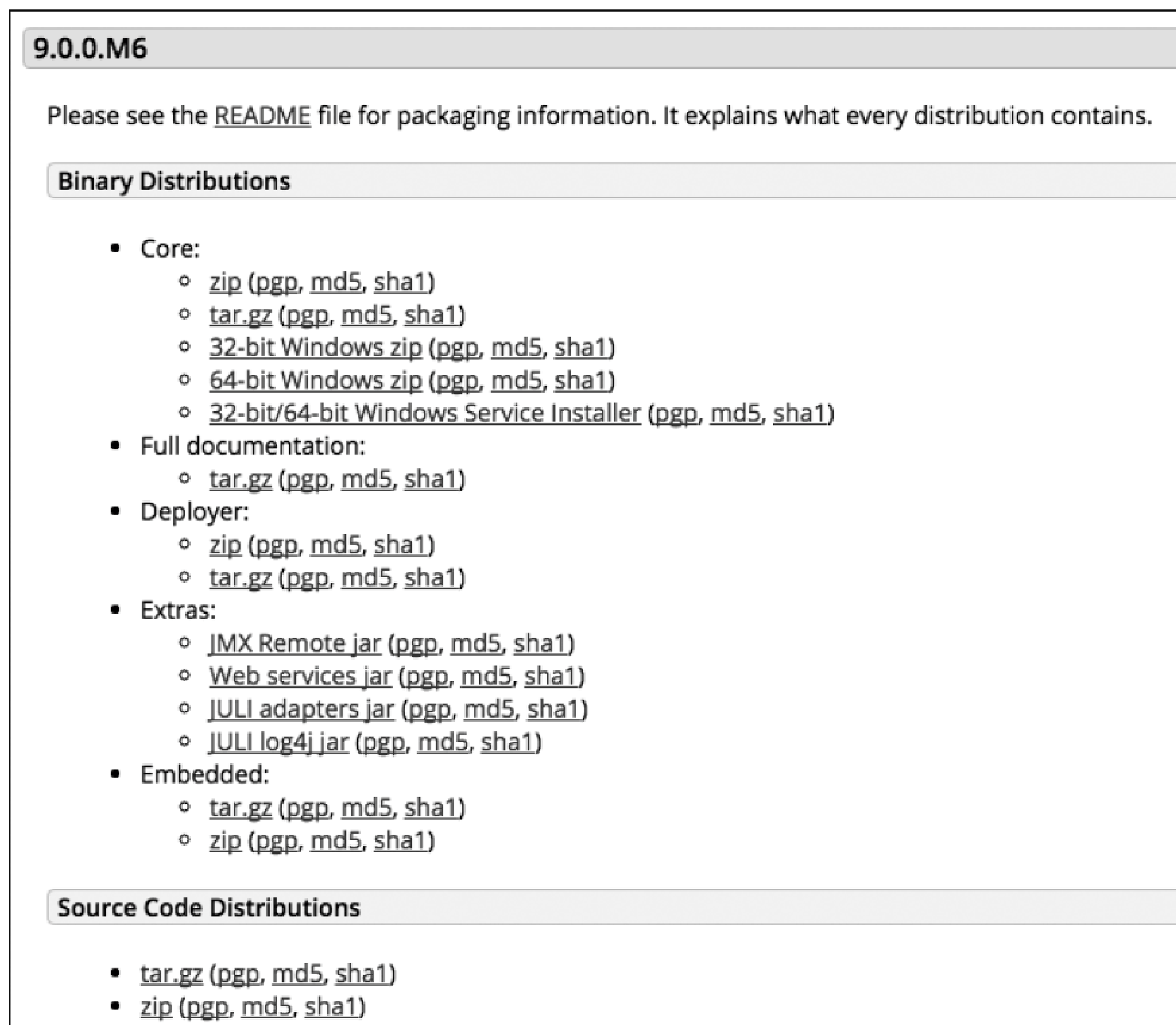


图 13-2　选择下载 Tomcat

13.3　编写简单的 Web 站点

本书使用 Eclipse IDE for Java EE Developers 进行 Web 站点的开发。Eclipse IDE for Java EE Developers 可以从 Eclipse 的官方网站（http://www.eclipse.org/downloads/）下载，下载之后解压压缩文件即可完成安装。

13.3.1　配置服务器运行环境

用 Eclipse IDE for Java EE Developers 编写 Web 站点的第一步是配置服务器运行环境。在 Eclipse 中选择 Window→Preferences 命令，打开偏好设置对话框，如图 13-3(a)所示。

(a)　　(b)

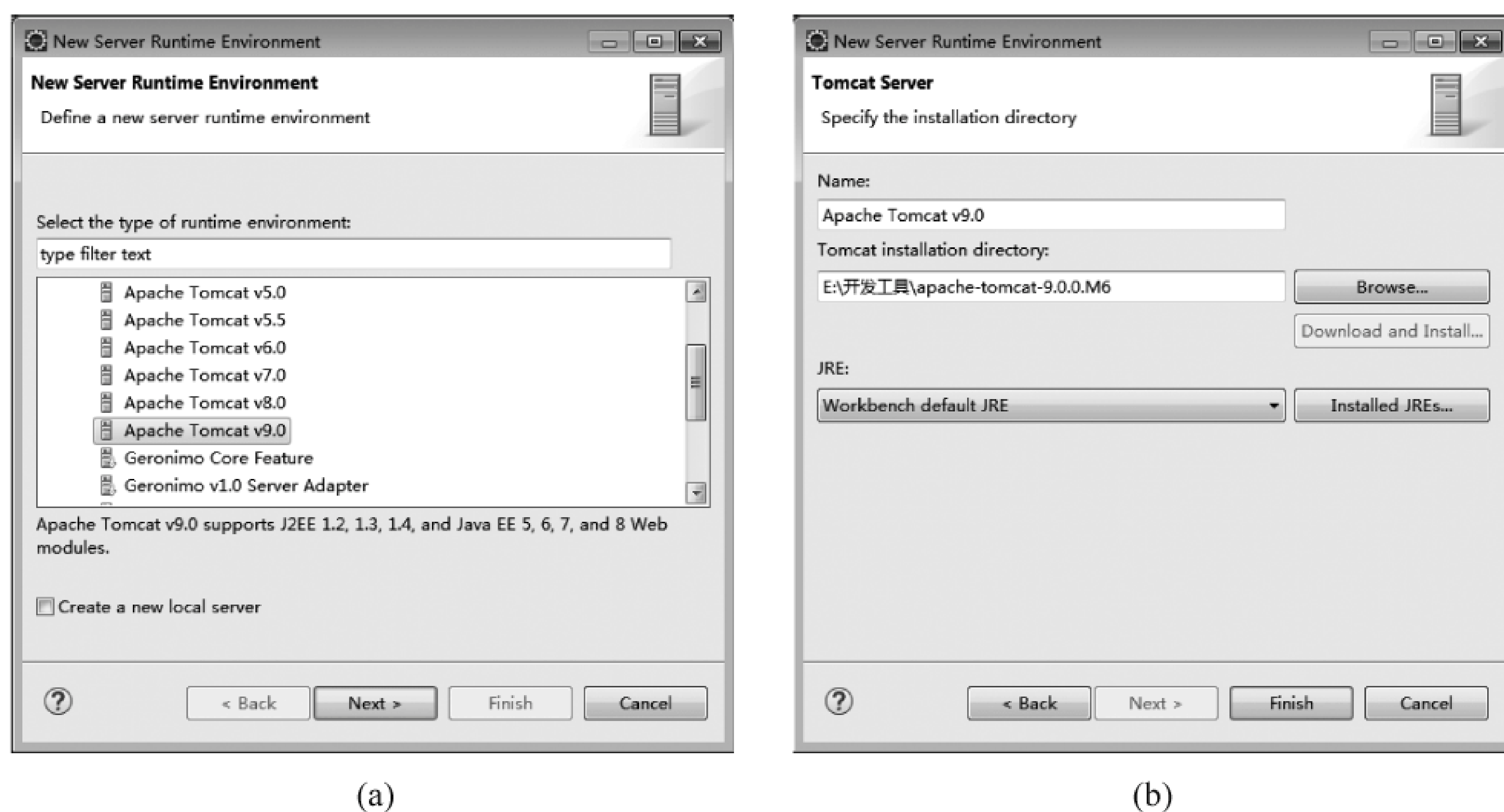
图 13-3　配置服务器运行环境

选择对话框中的 Server→Runtime Environment 命令，就打开了配置服务器运行环境的对话框，如图 13-3(b)所示。

在图 13-3(b)所示对话框中，单击 Add 按键，弹出图 13-4(a)所示的对话框。在该对话框中，可以选择要配置环境的服务器名称。Eclipse 已经预先设置了一些常用的服务器，包括 Apache Tomcat、IBM Websphere、JBoss 等。为了配置 Tomcat 的运行环境，这里选择 Apache Tomcat 9.0(如果使用其他的 Tomcat 版本，如 8.0，则需要选择对应的版本)。

选择之后弹出图 13-4(b)所示的对话框，在该对话框中输入 Tomcat 的安装位置(本书为 E:\开发工具\apache-tomcat-9.0.0.M6)，然后单击 Finish 按钮，Tomcat 的运行环境就配置好了。

Eclipse 的设置非常人性化，如果用户事先没有安装 Tomcat，在图 13-4(b)所示对话框中可以单击 Download and Install 按键，下载 Tomcat 9.0.0，并完成配置。



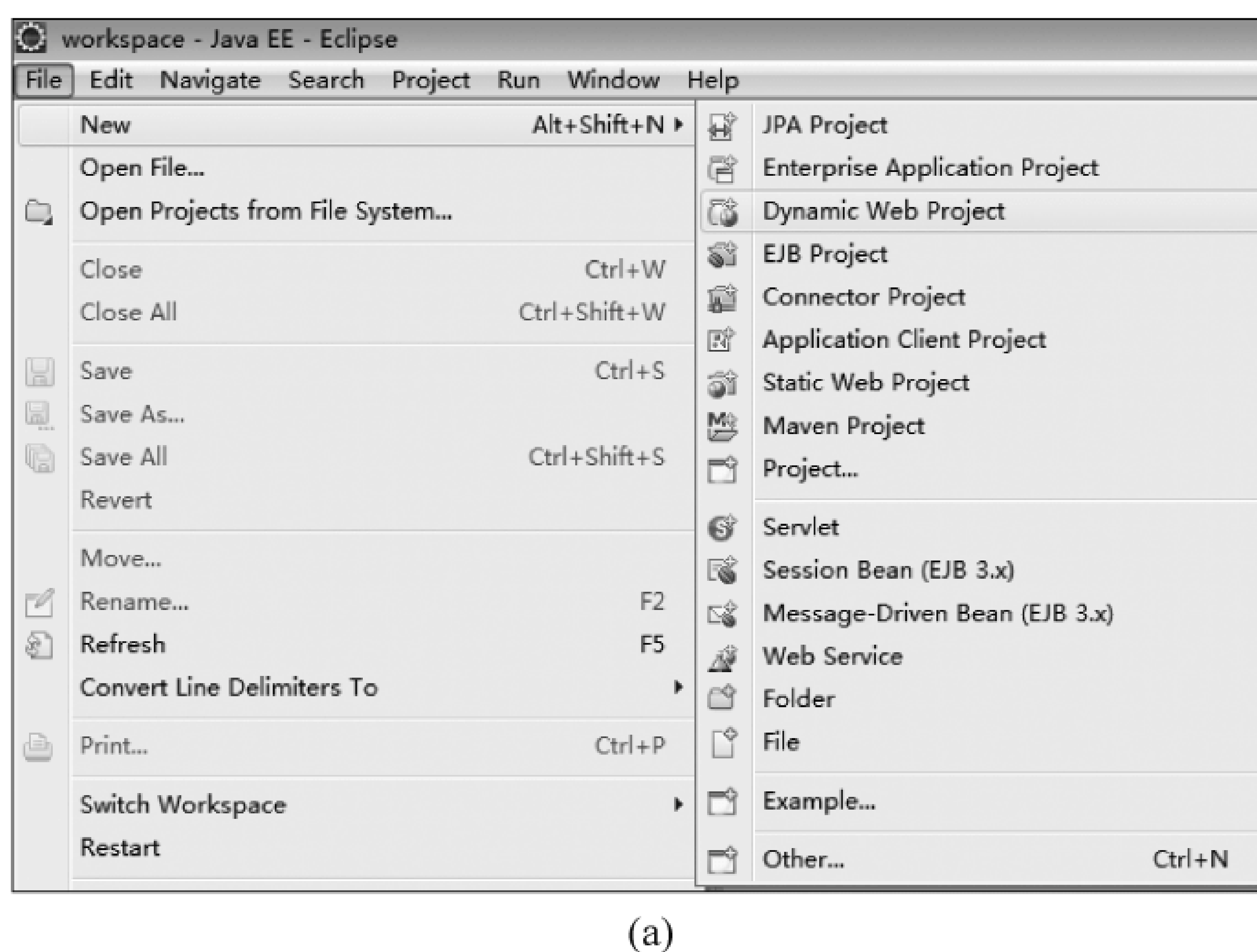
(a) (b)

图 13-4 配置 Tomcat 运行环境

13.3.2 新建动态 Web 工程

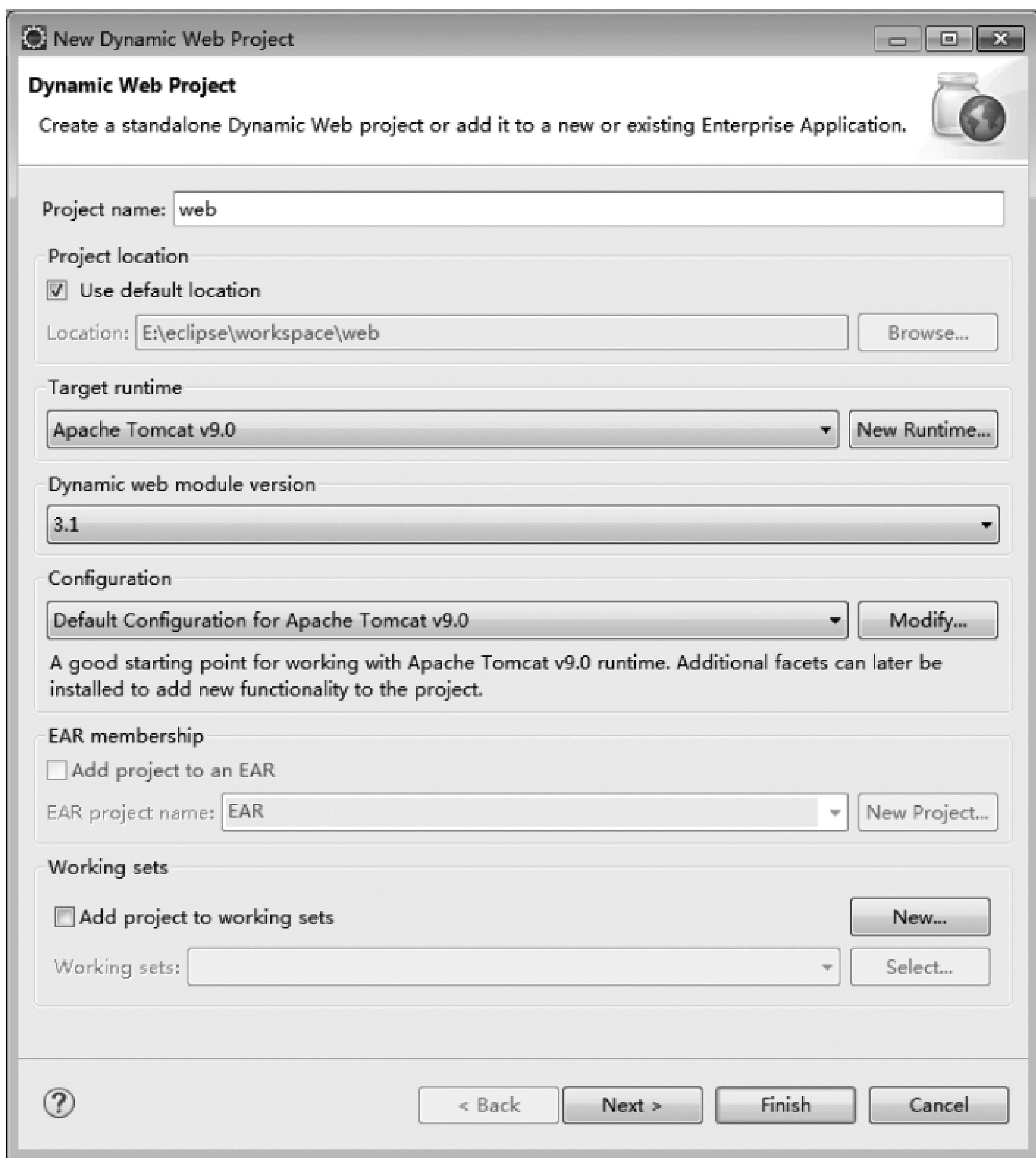
Tomcat 运行环境建立之后,下一步就是建立动态 Web 工程(Dynamic Web Project)。选择 File→New→Dynamic Web Project(如图 13-5(a)所示),打开新建动态 Web 工程的对话框,如图 13-5(b)所示。

输入工程名称(本书为 web),选择正确的 Target runtime。由于本书选择 Tomcat 9.0 作为 Web 站点的运行环境,因此在 Target runtime 下拉列表框中,选择 Apache Tomcat v9.0。对话框中的其他设置保持不变,单击 Finish 按钮,一个名称为 web 的动态 Web 工程就建好了。



(a)

图 13-5 新建动态 Web 工程



(b)

图 13-5 （续）

13.3.3 Web 工程的结构

新建的 Web 工程的结构如图 13-6 所示。

其中 Java Resources：src 目录放置 Java 源代码（主要是 Servlet 等 Java 类），WebContent 目录下放置 HTML、JSP、图片和视频等 Web 内容。

WebContent 目录下的 WEB-INF 目录非常重要，不能删除，也不能重命名。其下的 lib 目录用于放置 Web 工程可能用到的外部 Java 库文件。web. xml 文件是项目的配置文件，提供了 Error Pages、Filter Mappings、Filters、Listeners、References、Servlet Mappings、Servlets 和 Welcome Pages 等配置。Deployment Descriptor：web 是 Eclipse 提供的对 web. xml 文件的图形化编辑工具，Deployment Descriptor：web 中显示的信息与 web. xml 的文本信息是一一对应的。

13.3.4 新建 Servlet 和 JSP 程序

Servlet 和 JSP(Java Server Pages)是 Java Web 开发的主要技术。下面首先介绍如何创建一个 Servlet 程序。

在 Web 工程中右击目录 Java Resources：src，在弹出的快捷菜单中选择 New→Servlet（如图 13-7(a)所示），打开如图 13-7(b)所示的对话框。

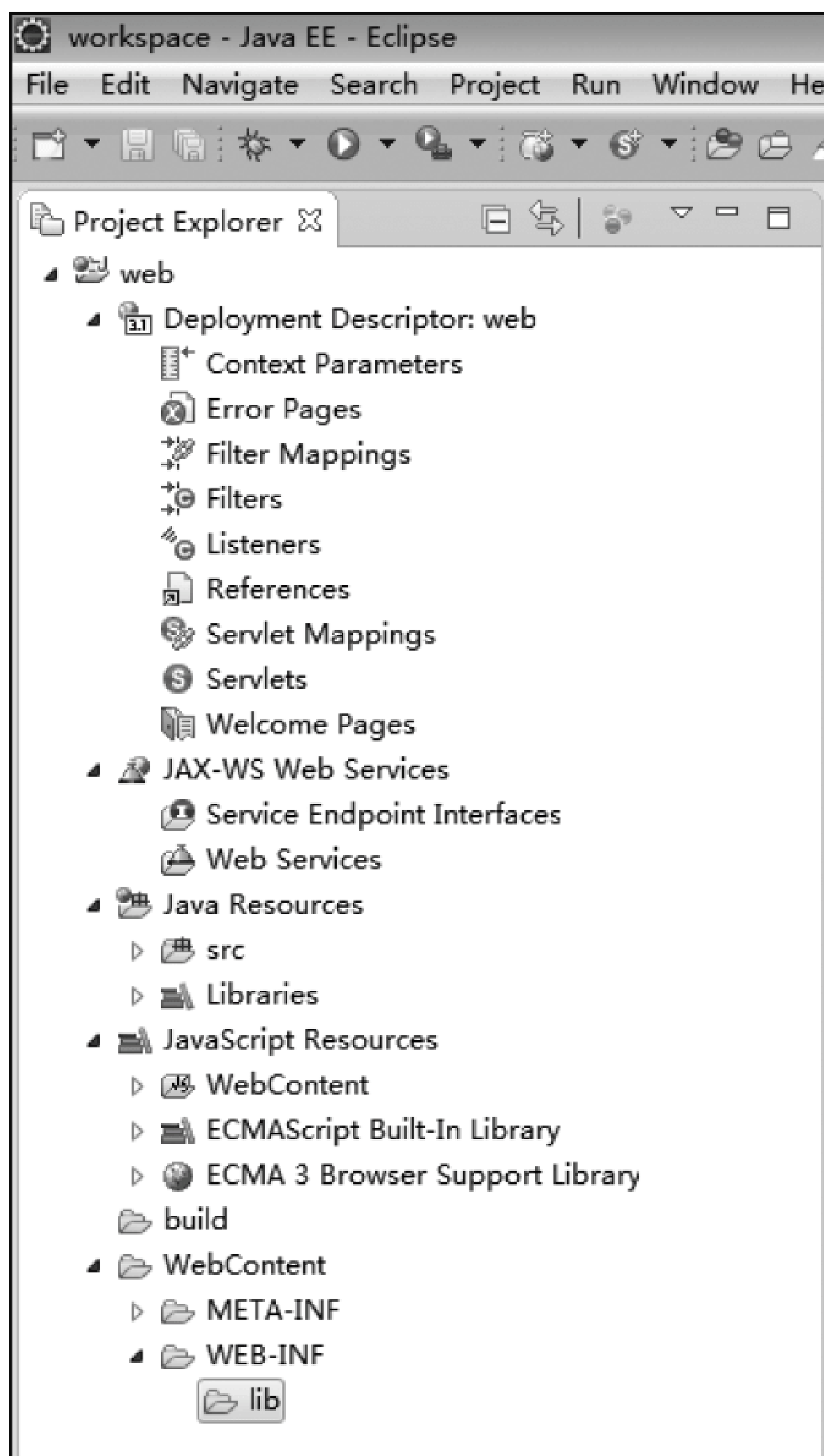


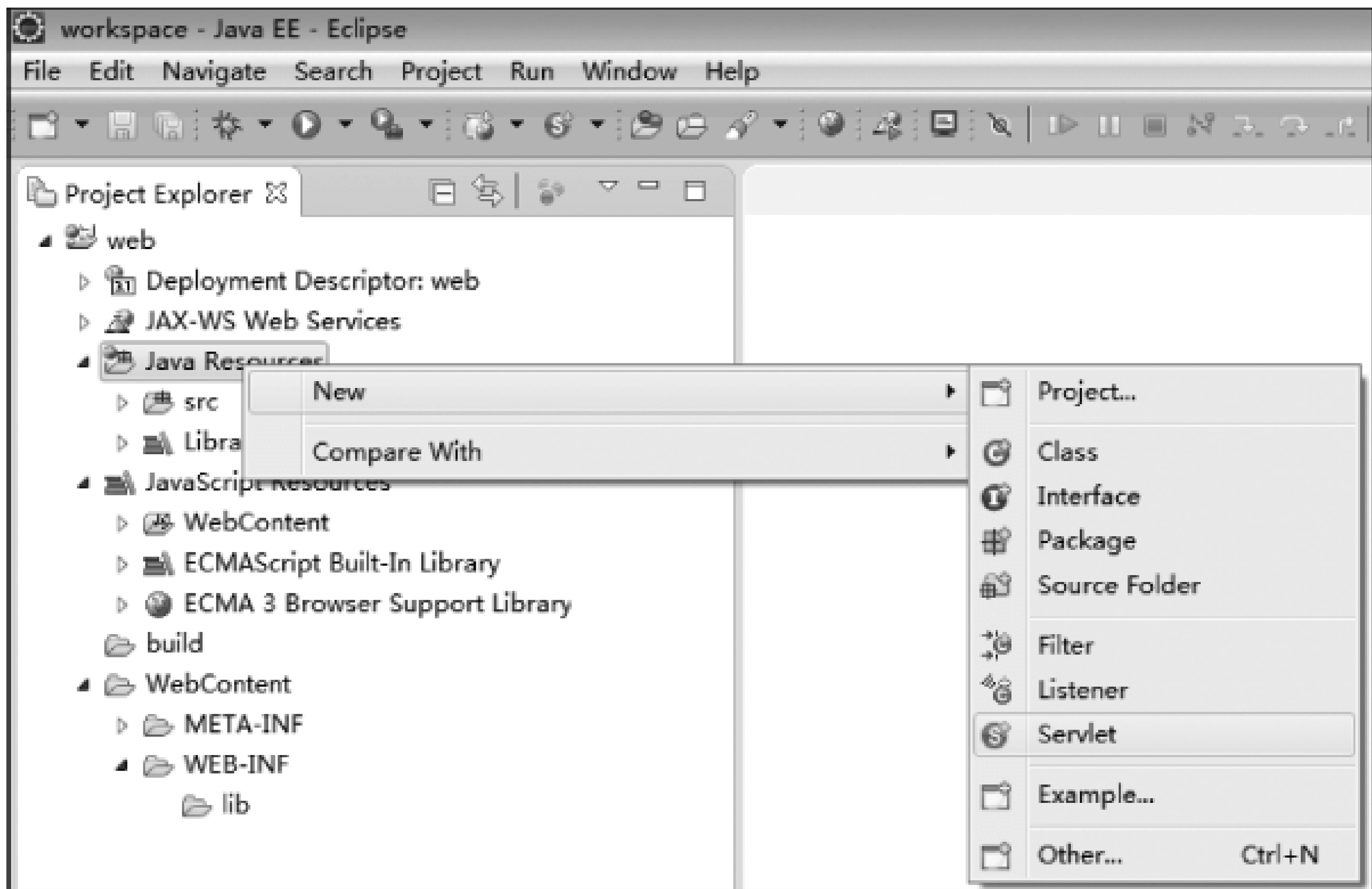
图 13-6　新建的 Web 工程的结构

所谓 Servlet,实际上是一个父类为 javax. servlet. http. HttpServlet 的 Java 类。因此,在图 13-7(b)所示的对话框中,要输入 Servlet 所对应的 Java 类的包名(Java Package)以及类名(Class name)。本书新建的 Servlet 的包名为 servlet,类名为 HelloServlet。Source folder 文本框指定了 HelloServlet 所对应源文件 HelloServlet. java 所在的目录(本书为/web/src),默认情况下不需要修改。

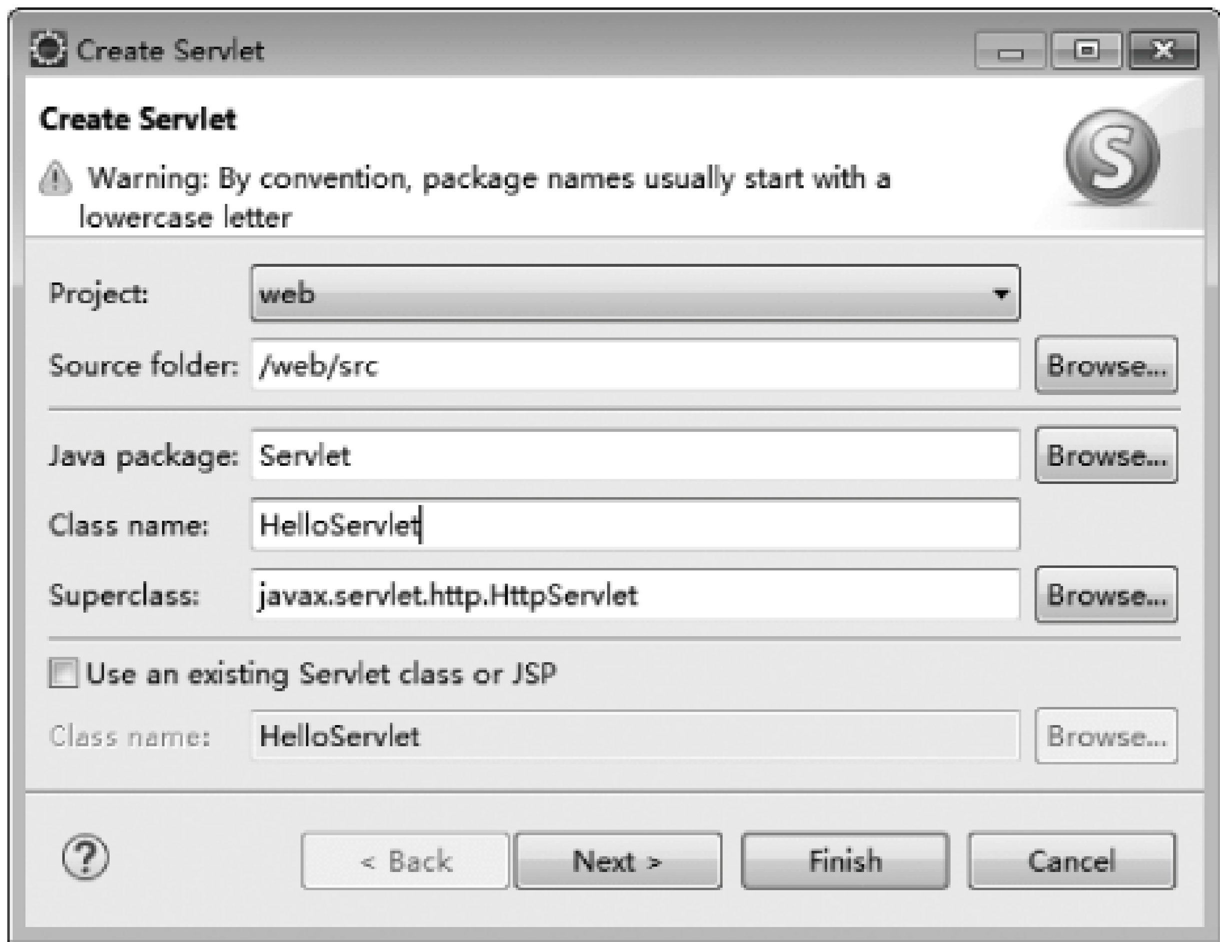
单击 Next 按钮打开如图 13-7(c)所示的对话框,在该窗口中设置 HelloServlet 的初始化参数和 URL 映射地址(URL Mappings)。所谓 URL 映射地址,是用户访问该 Servlet 的地址。同一个 Servlet 可以映射到多个 URL 地址,也可以映射为任意名称的 URL 地址。本书将 HelloServlet 映射为/helloservlet。

单击 Next 按钮打开如图 13-7(d)所示的对话框,在该窗口中设置 HelloServlet 所要重写的父类方法。默认选择重写 doGet 和 doPost 方法。doGet 方法用于处理 HTTP 协议中 Get 方式的请求,doPost 方法用于处理 HTTP 协议中 Post 方式的请求。Get 和 Post 是浏览器通过 HTTP 协议访问 Web 站点最常见的两种方式。最后单击 Finish 按钮。

选择 Java Resources: src 目录下 servlet 包中的 HelloServlet. Java 文件,用 Eclipse 的代码编辑器,编写如例 13.1 所示的 Java 代码,保存文件。HelloServlet 的创建就完成了。



(a)



(b)



(c)



(d)

图 13-7　新建 Servlet

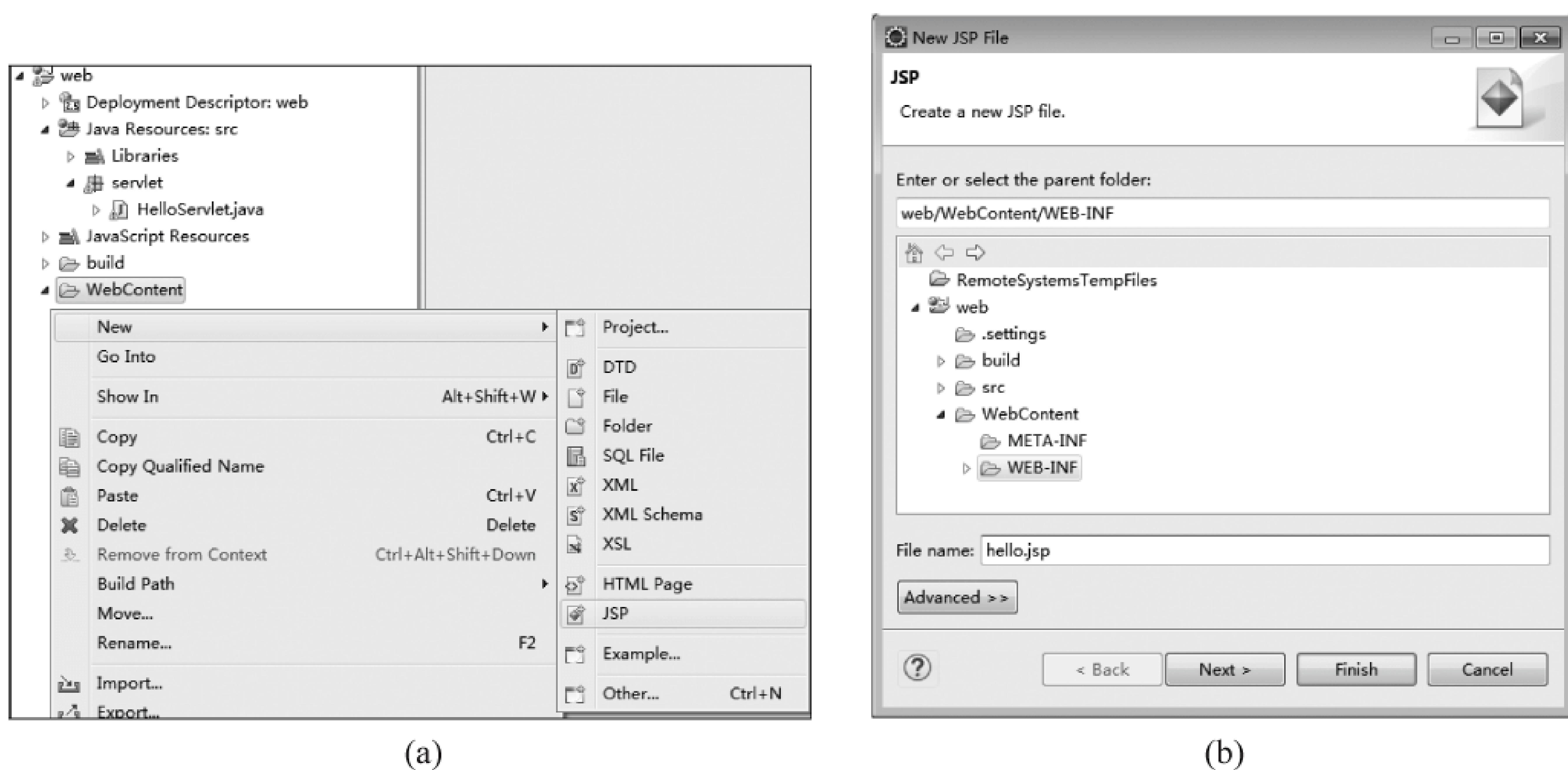
【例 13.1】　HelloServlet.Java。

```
package servlet;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class HelloServlet extends HttpServlet {
    public HelloServlet() {
        super();
    }
    //处理 Get 请求
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        //设置编码格式为 GB18030,否则无法正确显示中文
        response.setCharacterEncoding("GB18030");
```

```
            PrintWriter writer=response.getWriter();
            writer.println("Hello World! <br>");
            writer.println("这是 Servlet 输出的信息。");
            writer.close();
        }
        //处理 Post 请求
        protected void doPost(HttpServletRequest request,
                HttpServletResponse response) throws ServletException, IOException {
            //用同样的方法处理 Get 和 Post 请求
            doGet(request,response);
        }

}
```

下面介绍如何创建 JSP 程序。右击 WebContent 文件夹,在弹出的快捷菜单中选择 New→JSP 命令(如图 13-8(a)所示),打开 13-8(b)所示的对话框,在该对话框中输入 JSP 的文件名称(本书为 hello.jsp),单击 Finish 按钮就创建了一个名称为 hello.jsp 的 JSP 文件。



(a) (b)

图 13-8 新建 JSP

用编辑器打开 hello.jsp,输入如例 13.2 所示的代码,保存文件,JSP 程序的创建就完成了。

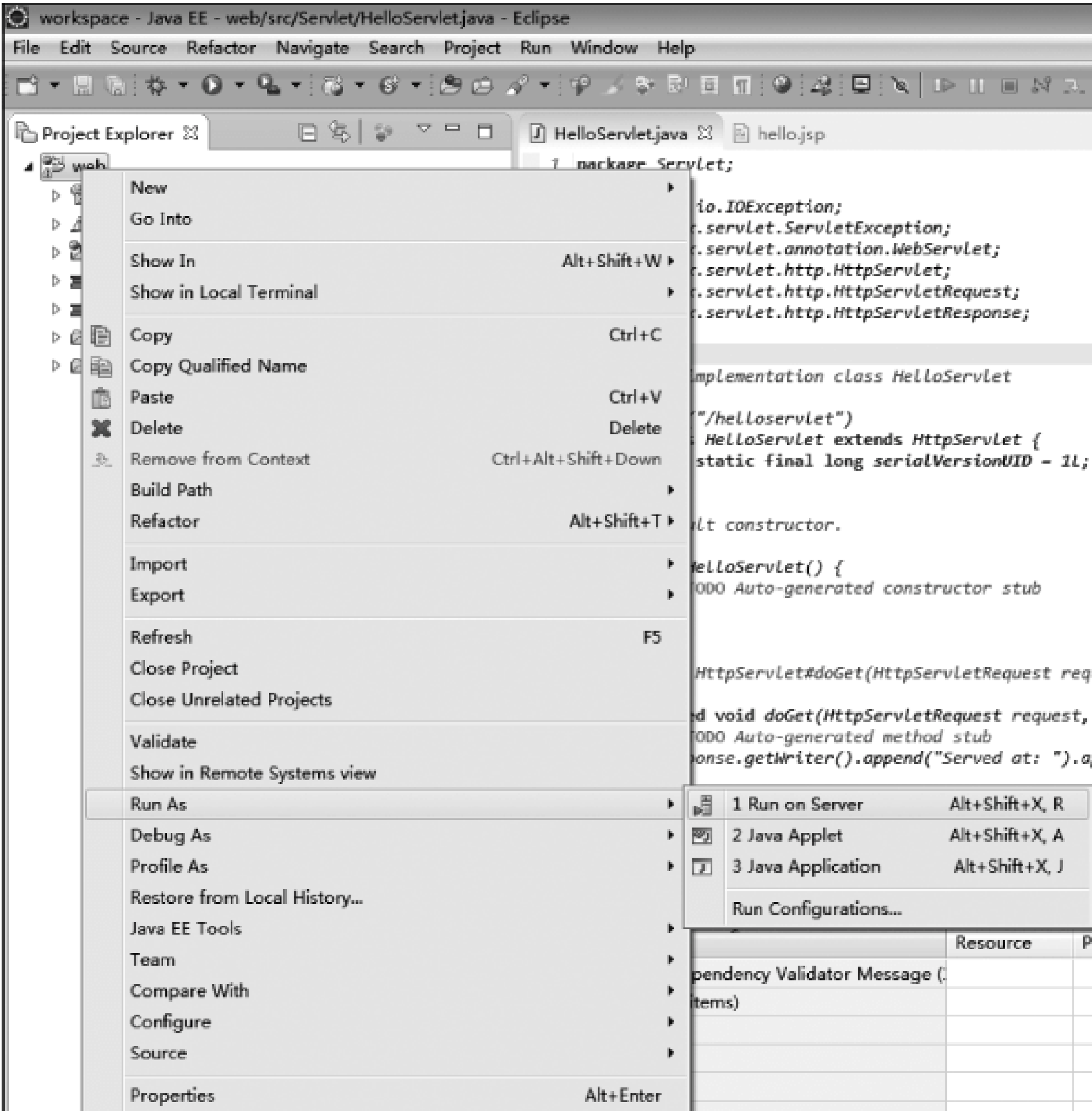
【例 13.2】 hello.jsp。

```
<%@ page language="java" pageEncoding="GB18030 "%>
<html>
<head>
<title>JSP</title>
</head>
<body>
Hello World! <br>
这是 JSP 输出的信息。
</body>
</html>
```

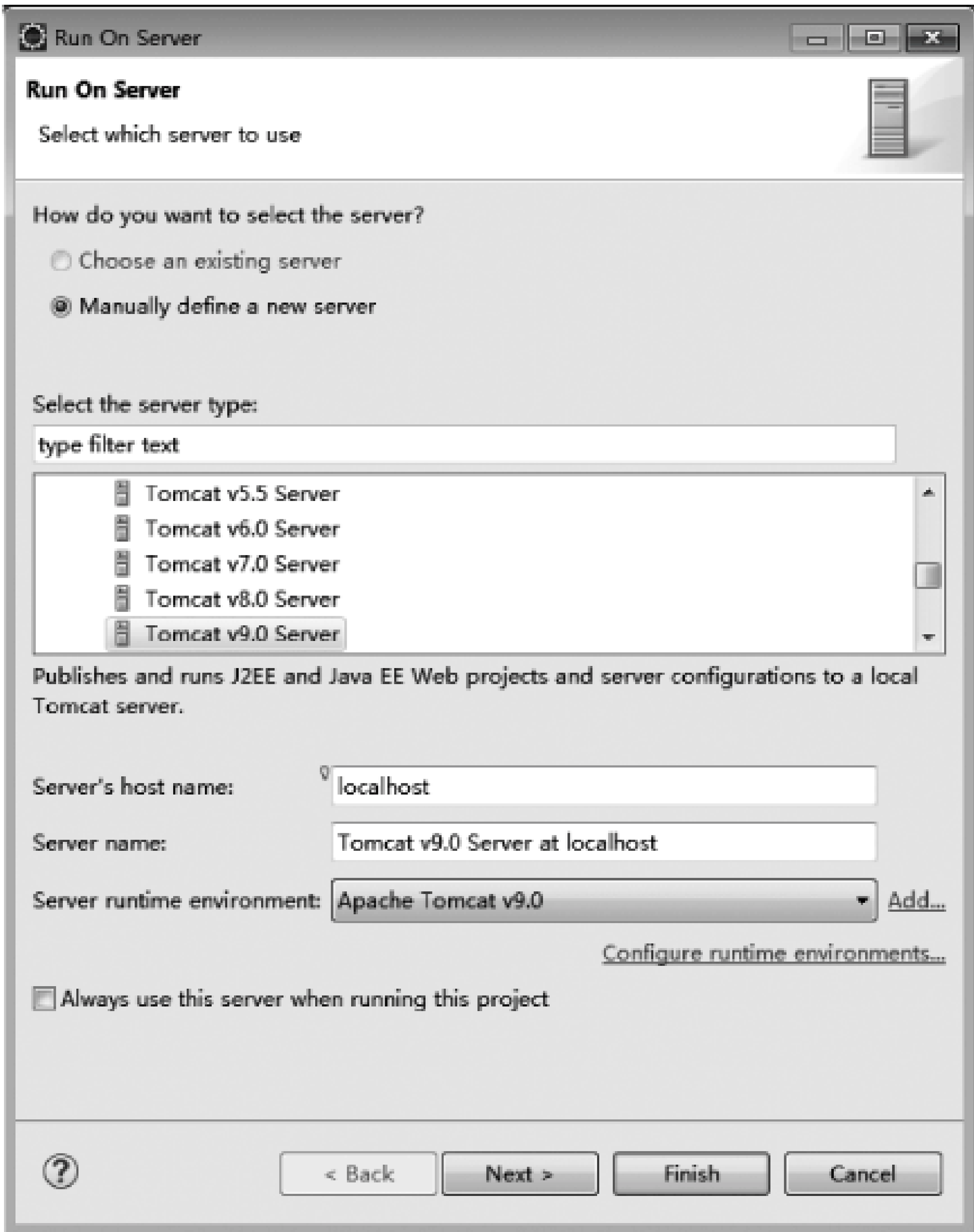
13.4 运行 Web 站点

第一次运行 Web 站点需要一些额外的设置。首先选中 Web 工程，单击工具栏上的“运行”按钮，在弹出的下拉列表中选择 Run As→Run on Server（如图 13-9(a)所示），打开如图 13-9(b)所示的对话框。

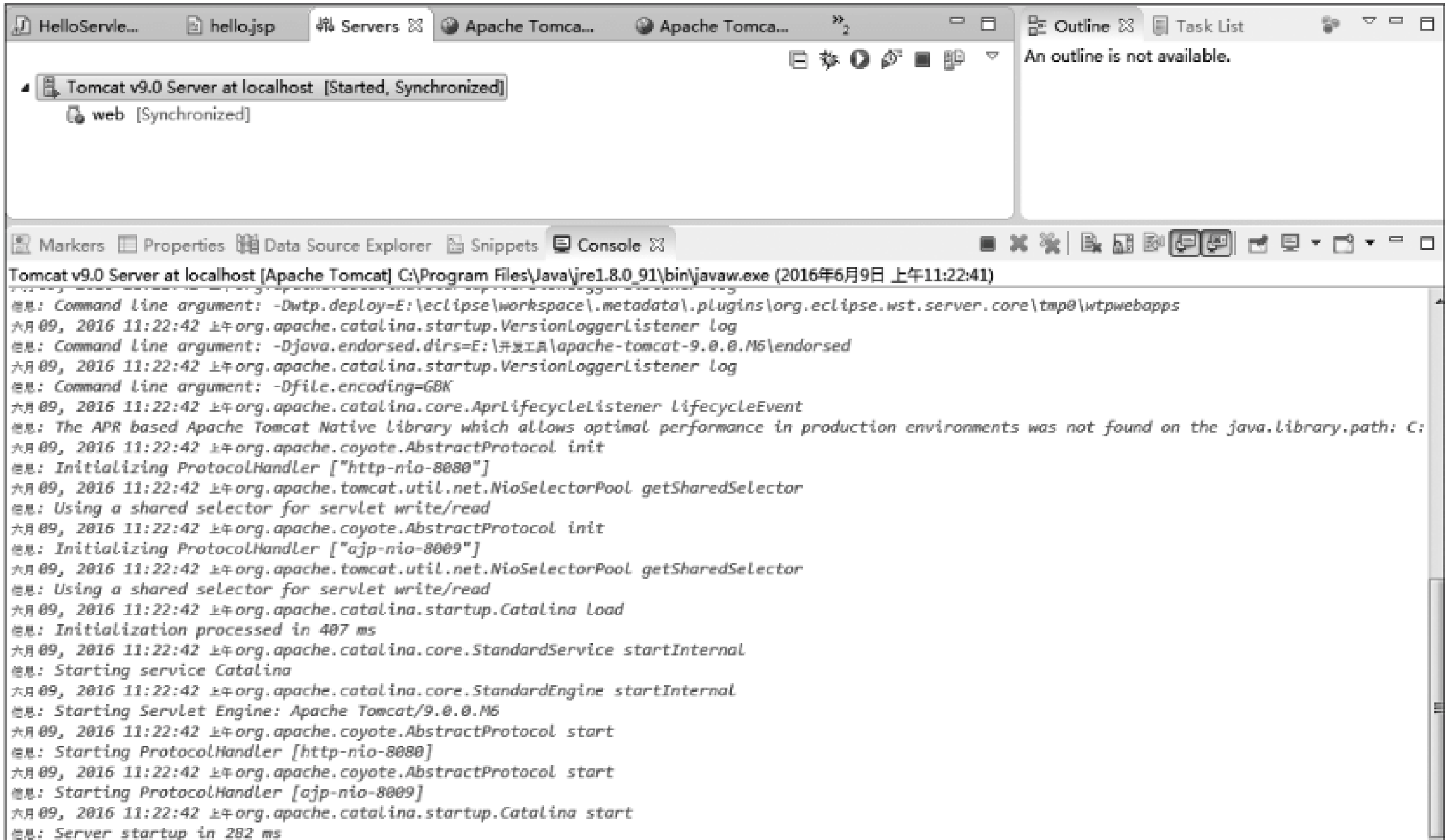
在如图 13-9(b)所示的对话框中，需要选择运行 Web 站点的服务器。由于本书使用 Tomcat 9.0.0 服务器，因此这里选择 Tomcat v9.0 Server。最后单击 Finish 按钮，Web 站点就启动了。Web 站点启动之后，可以在 Eclipse 界面下方的信息提示窗口，查看到服务器的运行状态和 Web 站点的启动情况，如图 13-9(c)所示。



(a)

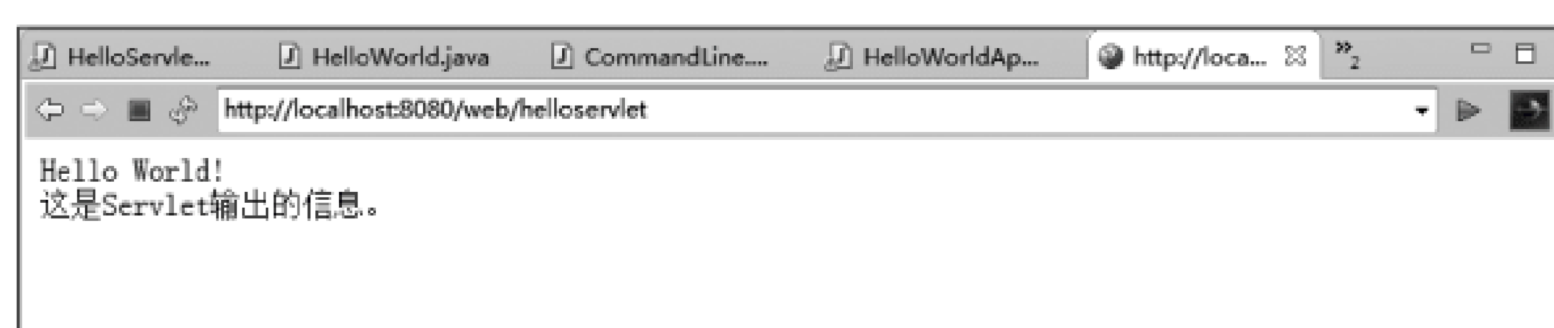


(b)



(c)

图 13-9 运行 Web 站点



(d)



(e)

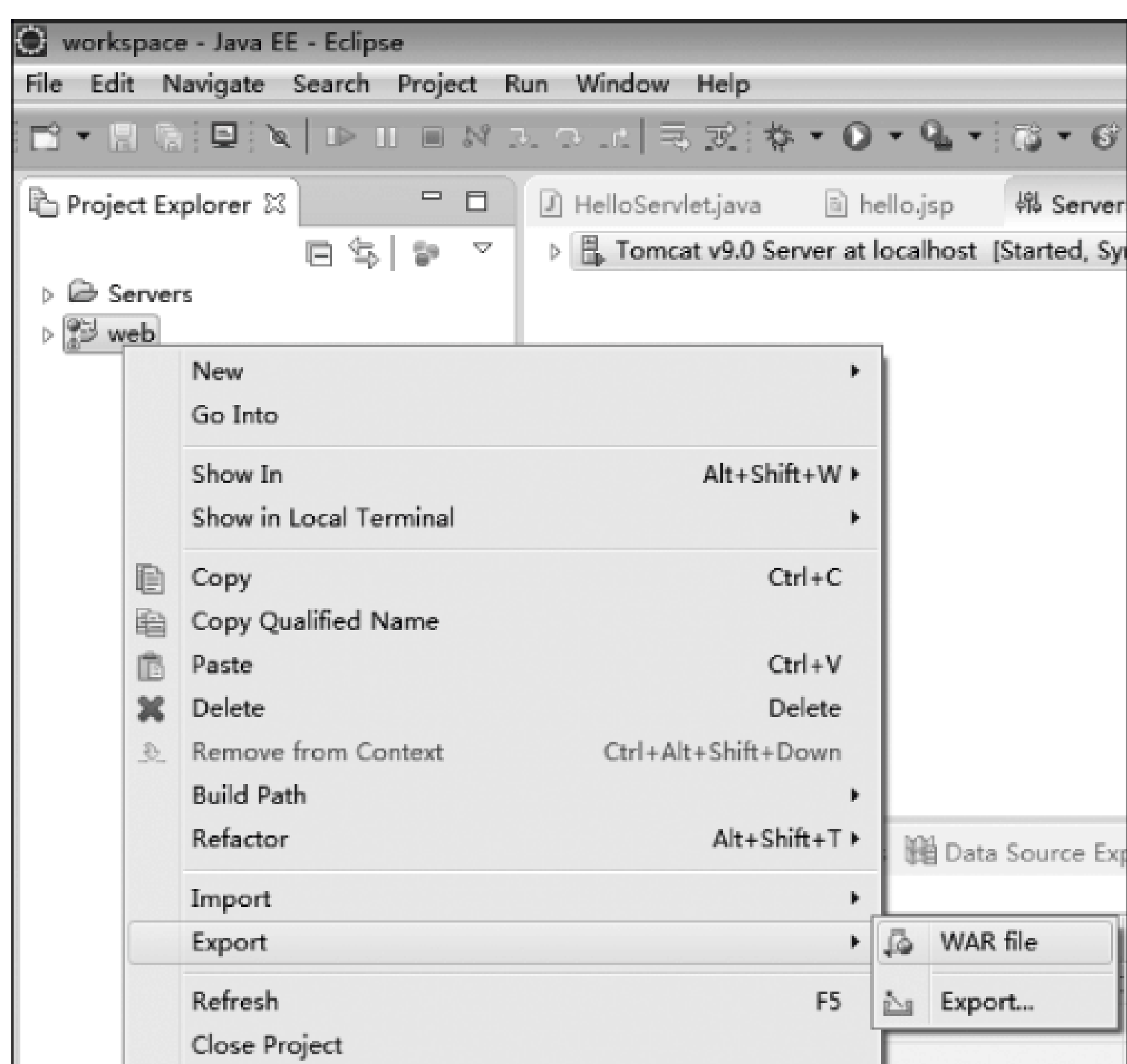
图 13-9 (续)

Web 站点启动之后,在打开的浏览器或者操作系统的浏览器(IE 或者 Firefox 等)的地址栏中输入 HelloServlet 对应的 URL 地址(http://localhost:8080/web/helloservlet),就可以查看 HelloServlet 的运行情况。在地址栏中输入 JSP 程序对应的 URL 地址(http://localhost:8080/web/hello.jsp),就可以查看 JSP 程序的运行情况。HelloServlet 和 hello.jsp 的运行情况如图 13-9(d)和图 13-9(e)所示。

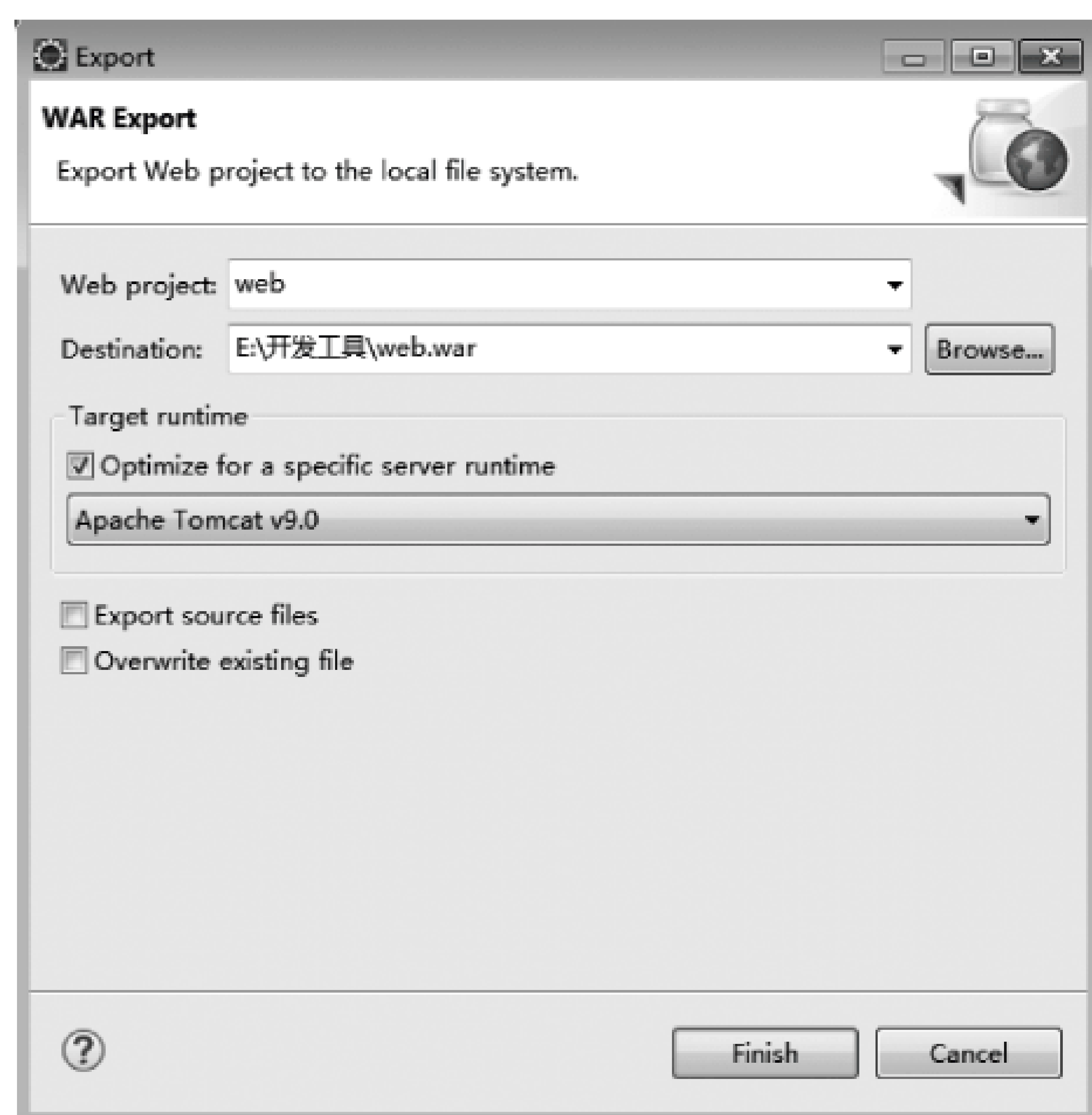
13.5 发布 Web 站点

在 Eclipse IDE for Java EE Developers 中运行的 Web 程序,只是为了便于开发调试。当一个 Web 程序完成开发之后,就需要部署到实际的服务器上运行。本节介绍如何将 13.4 节所示的 Web 程序发布到 Tomcat 中。

首先要导出 Web 站点。右击 Web 工程,在弹出的快捷菜单中选择 Export→WAR file 命令(如图 13-10(a)所示),打开如图 13-10(b)所示的对话框。在对话框中输入要导出的 WAR 文件的位置(Destination),这里为“E:\开发工具\web.war”。单击 Finish 按钮,就将整个 Web 站点导出为 web.war 文件。WAR 文件是一种压缩文件格式,该文件包含了 Web 站点的全部内容,可以用解压软件打开。



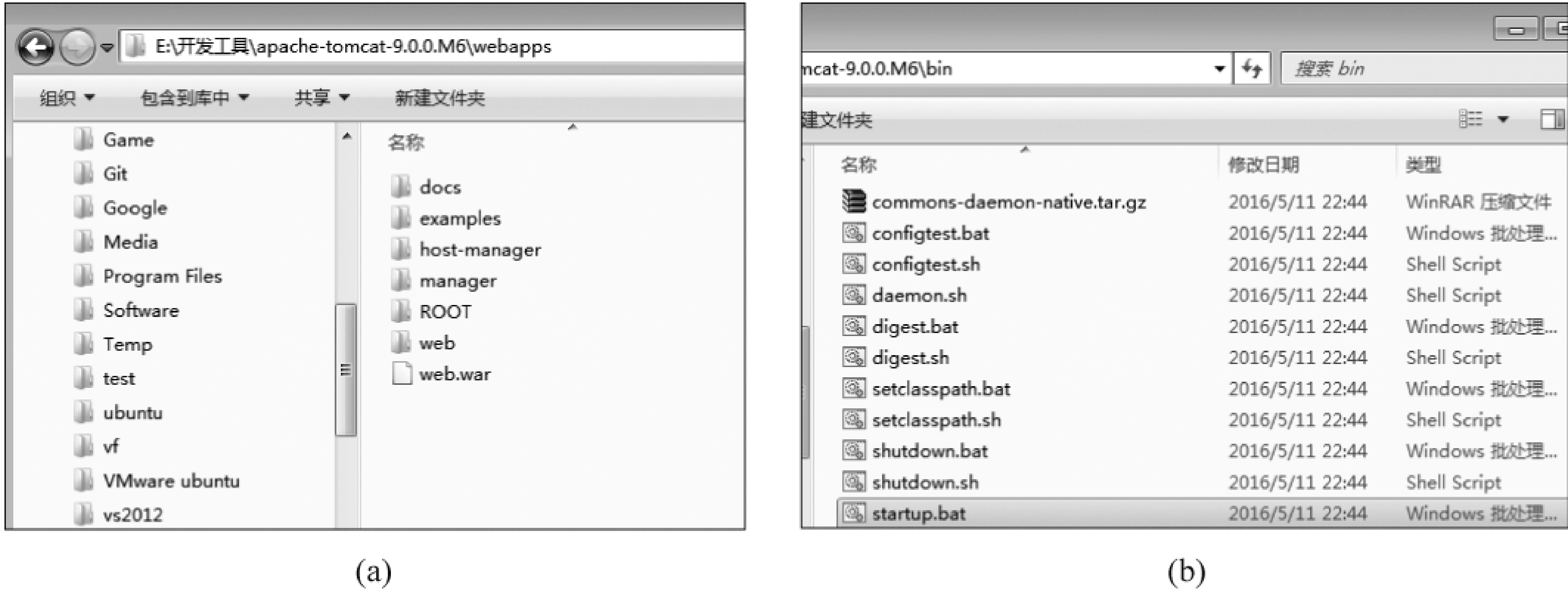
(a)



(b)

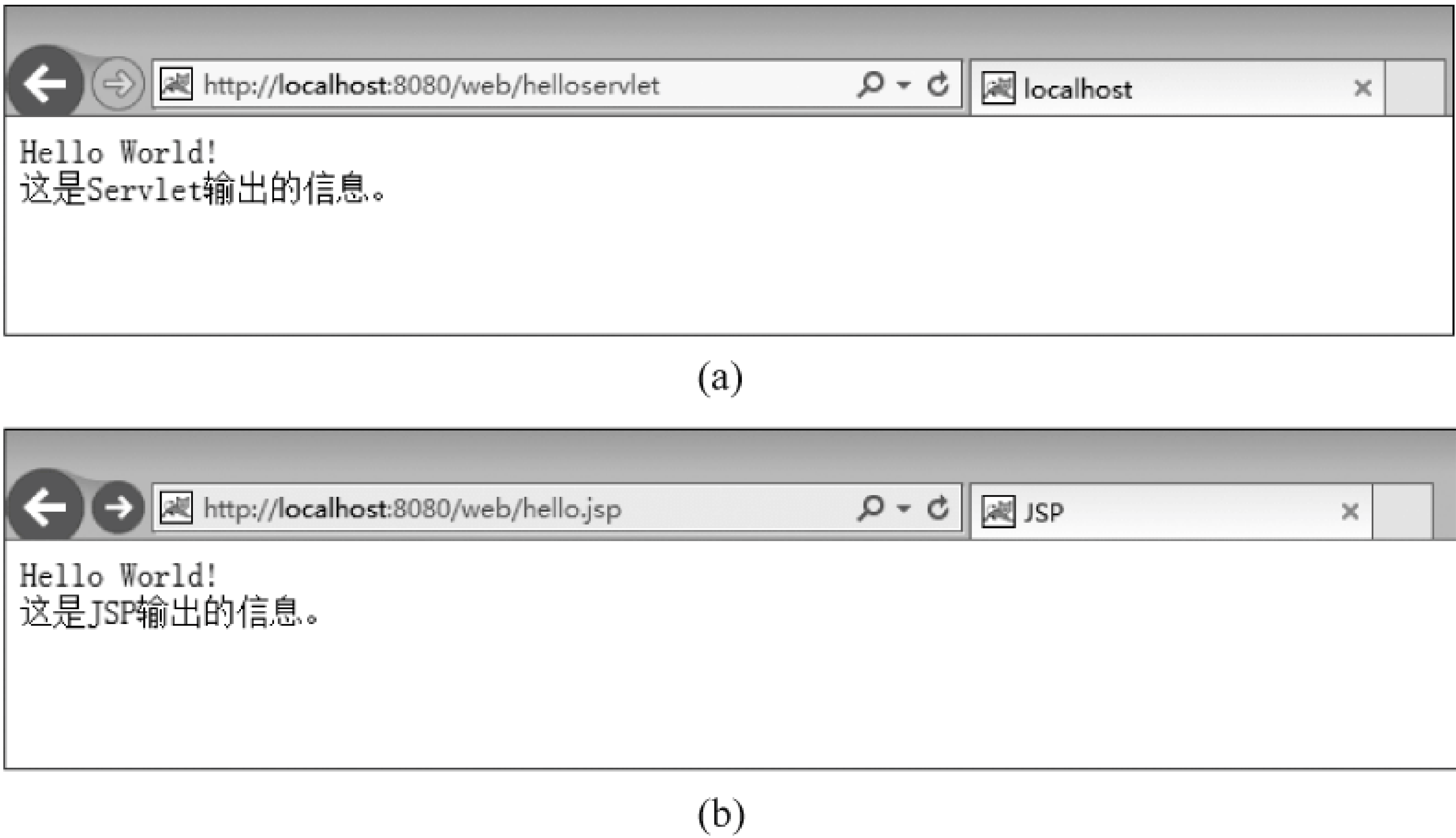
图 13-10 导出 Web 站点

下一步是部署 Web 站点。将导出的 web.war 文件复制到 tomcat 安装目录下的 webapps 文件夹(这里为“E:\开发工具\apache-tomcat-9.0.0.M6\webapps”),如图 13-11(a)所示。然后启动 Tomcat 服务器,即在 tomcat 安装目录下的 bin 文件夹下运行 startup.bat 程序,如图 13-11(a)所示。Tomcat 服务器启动后,在 IE 浏览器中查看 HelloServlet 和 hello.jsp 得到如图 13-12 所示的结果。



(a) (b)

图 13-11 部署 Web 站点



(a)

(b)

图 13-12 在 IE 中查看 Web 站点

习题与思考

1. 请结合网上的资料,学习 Tomcat 之外的 Web 服务器,如 jetty、resin 等。
2. Java 的 Web 服务器(如 Tomcat)与 ASP 常用的服务器 IIS 有什么区别?
3. 请尝试在 Eclipse IDE for Java EE Developers 中配置其他的 Web 服务器,并运行 Servlet 和 JSP 程序。
4. 请尝试在 Tomcat 之外的服务器上发布 Web 站点。

第14章 JDBC 技术

本章学习目标

在软件开发中，往往会用到数据库，所以掌握 JDBC 数据库编程技术非常重要。本章主要讲述如何在应用中通过 JDBC 连接和访问数据库。通过本章的学习，应该重点掌握以下主要内容：

（1）什么是 JDBC 技术。

（2）使用 JDBC 连接数据库的方法。

（3）JDBC 常用接口的使用。

14.1 为什么需要 JDBC

应用程序离不开对数据的处理。出于方便高效的考虑，往往将数据存储在数据库中。这就要求应用程序能够访问和处理存储在数据库中的数据。为此，人们定义了应用程序与数据库之间进行操作的接口。JDBC 就是 Java 程序连接和存取数据库的应用程序接口。

【例 14.1】 对于一个商品管理系统，系统中的数据都存储在数据库中。这里编写系统中的一个功能模块，即通过软件从数据库中提取商品的基本信息。假设商品的基本信息存储在 testDB 数据库的 commodity 表中，其中包含商品编号、商品名称、单价和状态 4 个字段信息。这里数据库使用 JavaDB。

```
package code1401;

import java.sql.* ;

public class QueryCommodityList {
    public static void main(String[] args) {
        String url= "jdbc:derby:testDB;create= true";
        Connection con= null;
        Statement stmt= null;
        String query= "SELECT *  FROM commodity";
```

```
        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            con= DriverManager.getConnection(url);
            stmt= con.createStatement();

            ResultSet rs= stmt.executeQuery(query);

            System.out.println("商品信息列表：");
            System.out.println("编号\t名称\t\t\t价格\t状态");
            while(rs.next()) {
                int id= rs.getInt("id");
                String name= rs.getString("name");
                float price= rs.getFloat("price");
                String status= rs.getString("status");
                System.out.println(id+ "\t"+ name+ "\t"+ price+ "\t"
                        + status);
            }
        } catch (ClassNotFoundException e) {
            System.err.print("类没有找到异常：");
            System.err.println(e.getMessage());
        } catch (SQLException e) {
            System.err.println("SQL 异常："+ e.getMessage());
        } finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                }
                stmt= null;
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                }
                con= null;
            }
        }
    }
}
```

程序可能的运行结果如图 14-1 所示。

该问题的求解涉及以下两个关键点：

（1）查询数据库中的商品信息用到了 SQL 语句 SELECT，实现分析参见 14.2 节。

（2）程序通过 JDBC 访问数据库的基本过程，关于 JDBC 的操作参见 14.4 节。

```
<terminated> QueryCommodityList [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (2016年6月26日 下午10:09:39)
编号      名称                  价格      状态
1         联想笔记本电脑G450L-TTH      2699.0    停产
2         惠普笔记本电脑CQ515   2950.0   在售
3         索尼笔记本电脑VPCEA27EC      5259.0    在售
```

图 14-1 例 14.1 程序可能的运行结果

14.2 数据库和常用的 SQL 语句

数据库管理系统(Database Management System, DBMS)是一个软件系统,它具有存储、检索和修改数据的功能。目前,主流的数据库管理系统是关系型数据库,包括 Oracle、Sybase、Microsoft SQL Server、DB2、MySQL 等产品。SQL(Structured Query Language,结构化查询语言)是目前使用的最为广泛的关系数据库语句。本节将对数据库技术和常用的 SQL 语句进行简要的介绍。

1. 创建、删除数据库

创建数据库的语法格式如下:

```
CREATE DATABASE [database_name]
```

对于不同的数据库,有不同的选项,但是每个数据库都会支持以上的语句。执行上面的语句时,数据库管理系统会使用默认值创建一个名为 database_name 的数据库。

删除一个数据库的语法格式如下:

```
DROP DATABASE [database_name]
```

2. 创建、删除表

创建表的语法格式如下:

```
CREATE TABLE table_name(
    column1 DATATYPE [NOT NULL] [NOT NULL PRIMARY KEY],
    column2 DATATYPE [NOT NULL],
…
)
```

创建表,必须定义表名、列名、列的类型和列的宽度。

删除表的语法格式如下:

```
DROP TABLE tableName
```

3. 插入一条数据

在 SQL 语句中,INSERT 语句是用来向表中添加记录的。INSERT 语句的基本语法格式如下:

```
INSERT INTO tableName (column1, column2, …)
VALUES (value1, value2, …)
```

4. 在表中删除数据

要删除表中已经存在的一条或多条记录，应该使用 DELETE 语句。DELETE 语句可以使用 WHERE 语句来选择删除特定的记录。

DELETE 语句的基本用法格式如下：

```
DELETE FROM tableName [WHERE …]
```

5. 更新表中的数据

要修改表中已经存在的一条或多条记录，应该使用 UPDATE 语句。UPDATE 语句可以使用 WHERE 语句来选择更新特定的记录。

UPDATE 语句的基本语法格式如下：

```
UPDATE tableName SET column1=values1, column2=values2, … [WHERE …]
```

6. 查询表中的数据

数据库查询是数据库的核心操作。使用 SELECT 语句进行数据库的查询，该语句具有灵活的使用方式和丰富的功能。其一般语法格式如下：

```
SELECT [ALL | DISTINCT]目标列表达式[,目标列表达式]…
FROM 基本表或视图[,基本表或视图]…
[WHERE 条件表达式]
[GROUP BY 列名 1 [HAVING 内部函数表达式]]
[ORDER BY 列名 2 [ASC | DESC]]
```

其中，SELECT 子句用于列出查询结果中的属性列，FROM 子句用于列出表达式求值中需要扫描的关系；WHERE 子句实现对记录的筛选；GROUP BY 子句根据指定列名对结果集中的元组进行分组；HAVING 子句对分组进行过滤，它一般与 GROUP BY 子句配合使用；ORDER BY 子句对结果集按指定列名进行排序，ASC 是升序，DESC 是降序。

SELECT 语句的含义是：根据 WHERE 子句的条件表达式，从 FROM 子句指定的基本表或者视图中找出满足条件的记录，再按 SELECT 子句中的目标列选出记录中的属性值形成结果集。如果有 GROUP BY 子句，则将结果按"列名 1"的值进行分组。如果有 ORDER BY 子句，则结果集还要按照"列名 2"的值的升序或者降序排序。

7. 条件子句

(1) WHERE 语句：使用 WHERE 语句可以选择满足条件的特定的记录。在使用 WHERE 语句时应注意：若列的数据类型为数字型，则不需要用引号；若列的数据类型为字符型，则需要用单引号把字符串括起来。

(2) IN 和 NOT IN：选择列值与值列表中某一个值相等的相关行信息；相反，NOT IN 则选择那些不在列表中的记录。

(3) BETWEEN…AND 和 NOT BETWEEN…AND：选择列值在某个范围或者不在该范围的记录。

(4) LIKE 和 NOT LIKE：用于查找字符串的匹配。通配符%匹配任意长度的字符串，而_只匹配一个字符。

(5) IS NULL 和 IS NOT NULL：用于查找列值为空值或非空值的记录。

(6) 逻辑运算 AND 和 OR：逻辑运算 AND 表示选择列值同时满足多个条件的记录；逻辑运算 OR 表示选择列值满足其中任意一个条件的记录。

14.3 JDBC 的结构

JDBC(Java Database Connectivity)是 Java 程序连接和存取数据库的应用程序接口，它为 Java 开发者使用数据库提供了统一的操作方式，JDBC 由一组 Java 类和接口组成。JDBC 使得开发人员可以使用纯 Java 的方式来连接数据库，并进行操作。

JDBC 由两层组成，上面一层是 JDBC API，下面一层是 JDBC 驱动程序 API，其功能结构如图 14-2 所示。JDBC API 负责与 JDBC 驱动程序管理器 API 进行通信，将各个不同的 SQL 语句发送给它。驱动程序管理器 API(对程序员是透明的)与实际连接到数据库的各个第三方驱动程序进行通信，并且返回查询的信息，或者执行由查询规定的操作。下面对各个部分分别进行说明。

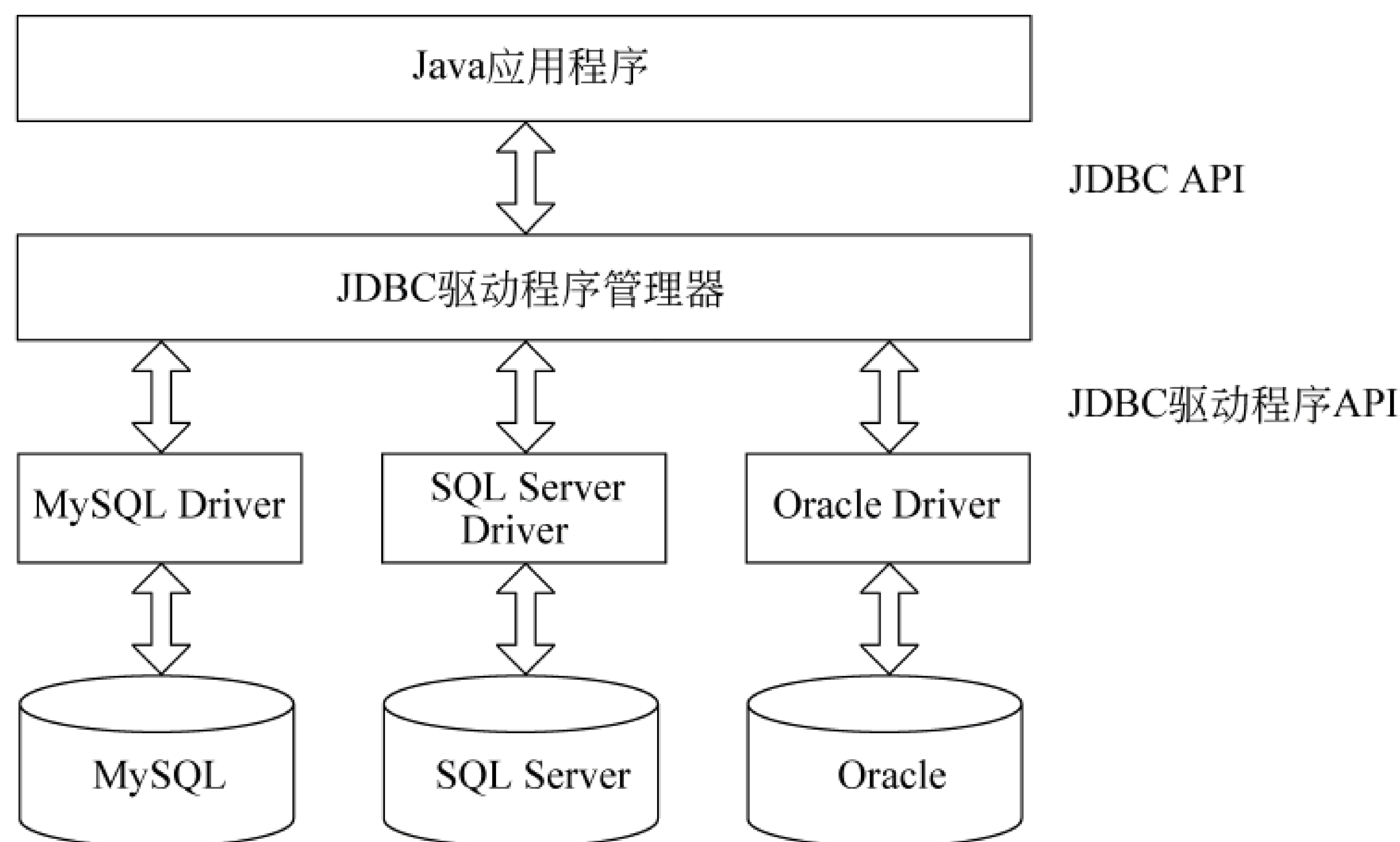


图 14-2　JDBC 功能结构图

1. Java 应用程序

Java 程序包括应用程序、Servlet 等，这些类型的程序都可以利用 JDBC 方法完成对数据库的访问和操作。

2. JDBC 驱动程序管理器

JDBC 驱动程序管理器能够动态地管理和维护数据库查询所需要的所有厂商或第三方所提供的驱动程序对象，实现 Java 任务与特定驱动程序的连接，从而体现 JDBC 与平台无关这一特点。它完成的主要任务有：为特定的数据库选择驱动程序；处理 JDBC 初始化调用；为每个驱动程序提供 JDBC 功能的入口；为 JDBC 调用执行参数等。

3. 驱动程序

这里的驱动程序(Driver)一般由数据库厂商或第三方提供，它由 JDBC 方法调用，向特定数据库发送 SQL 请求，并为 Java 程序获取结果。在必要的时候，驱动程序可以进行翻译或优化请求，使得 SQL 请求符合 DBMS 支持的语言。

JDBC 是独立于 DBMS 的，而每个数据库系统都有自己的协议与客户端通信，因此 JDBC 利用数据库驱动程序来使用这些数据库引擎。JDBC 驱动程序由数据库软件厂商和第三方提供，因此，使用的 DBMS 的不同，所需要的驱动程序也有所不同。

4. 数据库

这里的数据库是指 Java 程序需要的数据库以及数据库管理系统。

14.4 通过 JDBC 访问数据库

14.4.1 加载 JDBC 驱动程序

为了要连接数据库，必须要有相应数据库的 JDBC 驱动程序，并将驱动程序的.jar 文件加入到应用程序的 classpath 设置中。此后再在程序中通过 DriverManager 类加载 JDBC 驱动类。

DriverManager（驱动程序管理器）类是 JDBC 的管理层，作用于用户和驱动程序之间。DriverManager 类跟踪可用的驱动程序，并在数据库和相应驱动程序之间建立连接。另外，DriverManager 类也处理诸如驱动程序登录时间限制及登录和跟踪消息的显示等事务。

通过调用 Class.forName()方法将显式地加载驱动程序类。由于这与外部设置无关，因此推荐使用这种加载驱动程序的方法。例如，使用以下代码加载类 com.microsoft.sqlserver.jdbc.SQLServerDriver：

```
Class.forName("com.microsoft.sqlserver.jdbc.SQLServerDriver");
```

14.4.2 建立连接

当调用 DriverManager 类的 getConnection()方法时，DriverManager 类首先从它已加载的驱动程序池中找到一个可以接受该数据库 URL 的驱动程序，然后请求该驱动程序使用相关的数据库 URL 去连接到数据库中。于是，getConnection()方法将建立在 JDBC URL 中定义的数据库的 Connection 连接：

```
Connection con=DriverManager.getConnection(url, login, password);
```

JDBC 的 URL 的语法格式如下：

```
jdbc:<子协议>:<子名>
```

这里有 3 个部分，它们用冒号隔离。其中，jdbc 表示协议，它是唯一的，JDBC 只有这一种协议；子协议主要用于识别数据库驱动程序；子名属于专门的驱动程序，不同的专有驱动程序可以采用不同的实现。对于不同的数据库，厂商提供的驱动程序和连接的 URL 都不同，常见的数据库驱动程序和 URL 如表 14-1 所示。

getConnection()方法返回的 Connection 对象代表与数据库的连接。程序必须创建一个 Connection 类的实例，其中包括要连接数据库的信息。一个应用程序可与单个数据库有一个或多个连接，也可以与多个数据库有连接。

表 14-1 常见的数据库驱动程序和 URL

数据库名	驱动程序	URL
JDBC-ODBC	sun.jdbc.odbc.JdbcOdbcDriver	jdbc:odbc:[odbcsource]
Microsoft SQL Server 2008	com.microsoft.sqlserver.jdbc.SQLServerDriver	jdbc:sqlserver://[ip]:[port];user=[user];password=[password]
MySQL	org.gjt.mm.mysql.Driver	jdbc:mysql:/ip/database,user,password
JavaDB	org.apache.derby.jdbc.EmbeddedDriver	jdbc:derby:testDB;

Connection 接口常用的方法有以下几种。

(1) close()方法:关闭到数据库的连接,在使用完连接后必须关闭,否则连接会保持一段比较长的时间,直到超时。

(2) createStatement()方法:创建一个 Statement 对象用于执行 SQL 语句。

(3) prepareStatement()方法:使用指定的 SQL 语句创建一个预处理语句,SQL 中的参数用?占位符表示。

14.4.3 执行 SQL 语句

连接一旦建立,就可用来向它所涉及的数据库传送 SQL 语句。JDBC 对可被发送的 SQL 语句类型不加任何限制,这就提供了很大的灵活性,即允许使用特定的数据库语句甚至非 SQL 语句。然而,它要求用户自己负责确保所涉及的数据库可以处理所发送的 SQL 语句,否则将产生错误。例如,如果某个应用程序试图向不支持存储程序的 DBMS 发送存储程序调用,就会失败并将抛出异常。

JDBC 通过 Statement 接口向数据库发送 SQL 语句。Statement 提供了许多方法,最常用的方法有以下几种。

(1) execute()方法:运行语句,返回是否有结果集。

(2) executeQuery()方法:运行查询语句,返回 ResultSet 对象。

(3) executeUpdate()方法:运行更新操作,返回更新的行数。

有 3 种 Statement 对象,它们都作为在给定连接上执行 SQL 语句的包容器:Statement、PreparedStatement(它从 Statement 继承而来)和 CallableStatement(它从 PreparedStatement 继承而来)。Statement 对象用于执行不带参数的简单 SQL 语句,PreparedStatement 对象用于执行带或者不带参数的预编译 SQL 语句,CallableStatement 对象用于执行对数据库已存储过程的调用。

14.4.4 检索结果

SQL 语句发送以后,返回的结果通常存放在一个 ResultSet 类的对象中,ResultSet 对象可以看成是一个表,这个表中包含由 SQL 返回的列名和相应的值。

ResultSet 对象中维持了一个指向当前行的指针,通过 ResultSet.next()方法把当前的指针向下移动一行。最初它位于第一行前,因此第一次调用 next()方法将把指针置于第一行上,使它成为当前行。随着每次调用 next()方法导致指针向下移动,按照从上到下的次

序获取 ResultSet 行。

ResultSet 提供了检索不同类型字段的方法，最常用的方法如下。

(1) getString()方法：获得在数据库里是 varchar、char 等数据类型的数据。

(2) getFloat()方法：获得在数据库里是 float 数据类型的数据。

(3) getDouble()方法：获得在数据库里是 double 数据类型的数据。

(4) getBlob()方法：获得在数据库里是 blob(二进制大型对象)数据类型的数据。

(5) getClob()方法：获得在数据库里是 clob(字符串大型对象)数据类型的数据。

一系列的 getXXX 方法提供了获取当前行中某列值的途径，在每一行内，可按任意次序获取列值。

14.4.5 关闭连接

在对象使用完毕后，应当使用 close()方法解除与数据库的连接，并关闭数据库。例如：

```
con.close();
```

图 14-3 显示了一个用简单的 JDBC 模型进行连接、执行和获取数据的过程，其中我们只做了一次连接。实际上 DriverManager 一次可以有多个连接，而一个 Connection 可以执行多个 SQL 语句，图 14-4 所示为一个复杂的通过 JDBC API 访问 SQL 数据库的例子。

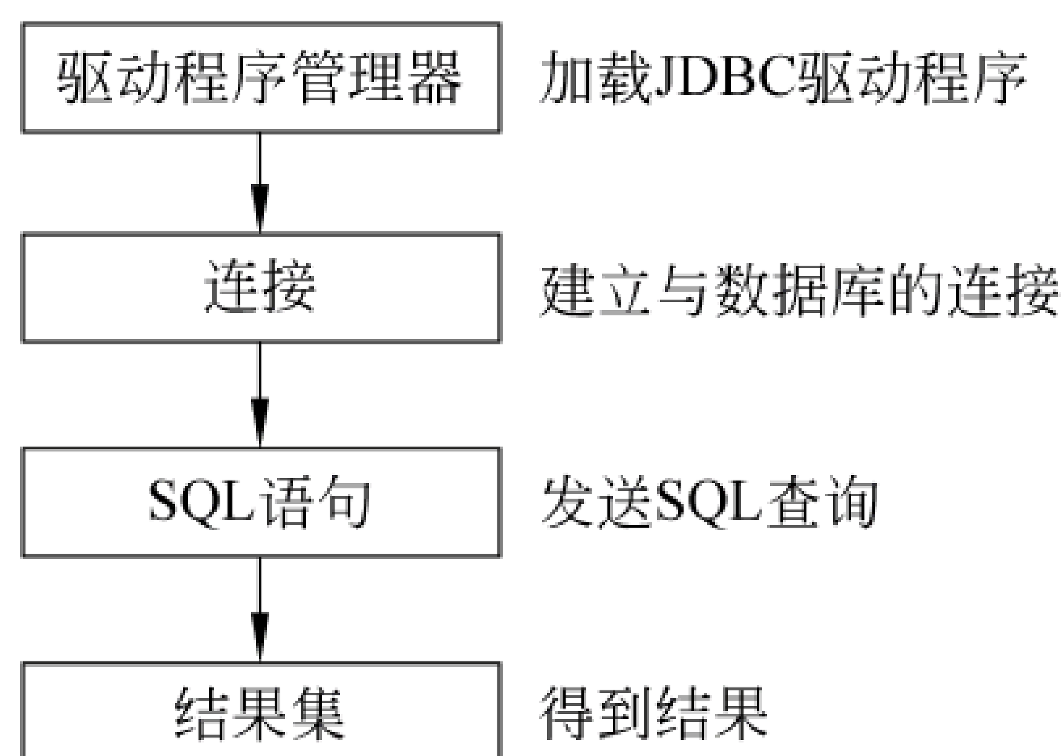


图 14-3　一个简单的通过 JDBC 访问数据库的过程

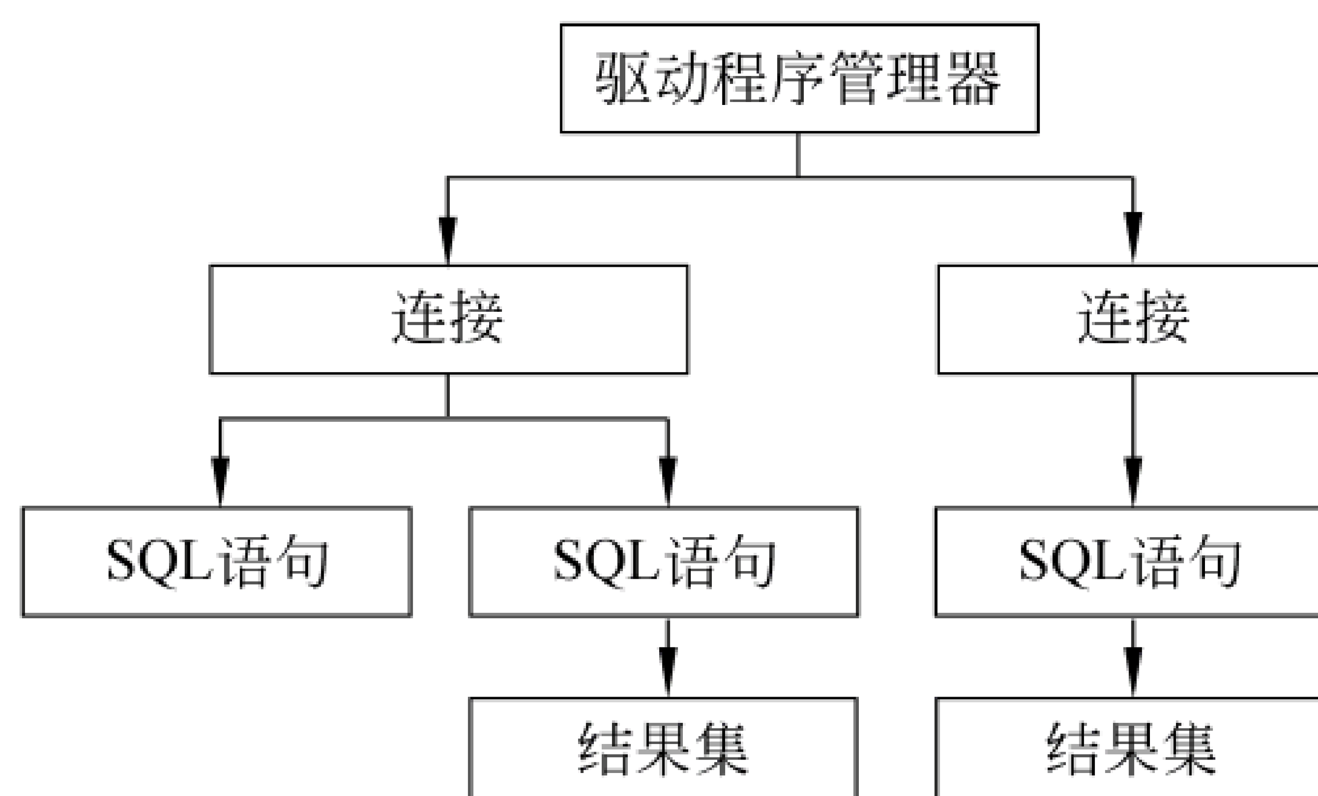


图 14-4　一个复杂的通过 JDBC API 访问 SQL 数据库的过程

14.4.6 实用案例

【例 14.2】 查询指定商品状态的 Java 应用程序。

```
package code1404;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.* ;

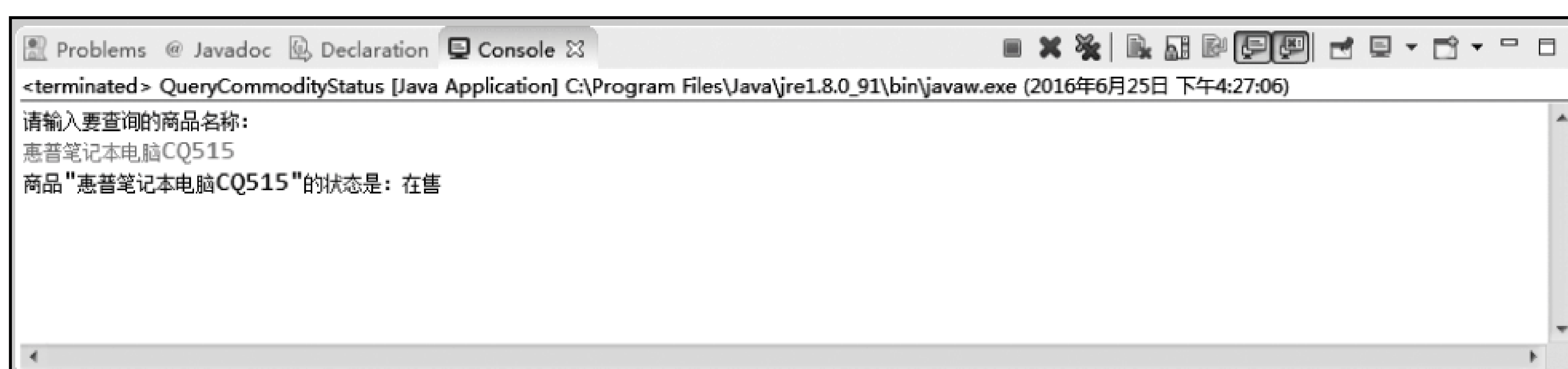
public class QueryCommodityStatus {
    public static void main(String[] args) {
        String url= "jdbc:derby:testDB;create= true";
        Connection con= null;
        PreparedStatement pstm= null;

        try {
//加载 JDBC 驱动程序
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            con= DriverManager.getConnection(url); //建立连接
            pstm= con.prepareStatement("SELECT status FROM commodity WHERE name=?"); //创建 PreparedStatement 语句

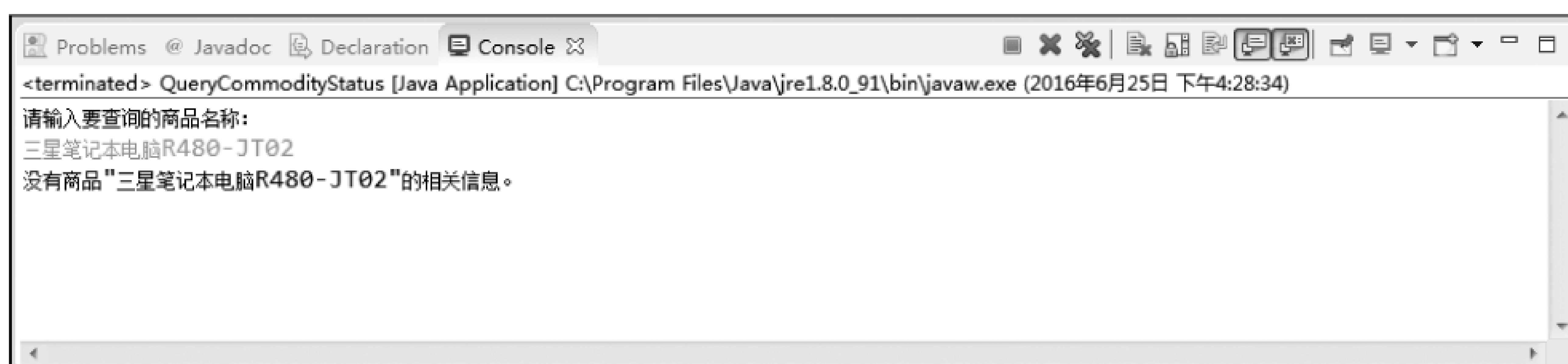
            System.out.println("请输入要查询的商品名称:");
            String inputName= (new BufferedReader(new InputStreamReader(
                    System.in))).readLine();
            pstm.setString(1, inputName);           //对 SQL 语句中的参数进行赋值
            ResultSet rs= pstm.executeQuery();      //执行 SQL 语句
            if (rs.next()) {
                String status= rs.getString("status");        //检索结果
                System.out.println("商品\""+ inputName+ "\"的状态是:"+ status);
            } else {
                System.out.println("没有商品\""+ inputName+ "\"的相关信息。");
            }
        } catch (ClassNotFoundException e) {
            System.err.print("类没有找到异常:");
            System.err.println(e.getMessage());
        } catch (SQLException e) {
            System.err.println("SQL 异常:"+ e.getMessage());
        } catch (IOException e) {
            System.err.println("IO 异常:"+ e.getMessage());
        } finally {
            if (pstm != null) {
                try {
                    pstm.close();                   //关闭 PreparedStatement 语句
                } catch (SQLException e) {
                }
                pstm= null;
            }
```

```
                if (con != null) {
                    try {
                        con.close();                    //关闭连接
                    } catch (SQLException e) {
                    }
                    con= null;
                }
            }
        }
}
```

程序可能的运行结果如图 14-5 所示。



(a)



(b)

图 14-5 例 14.2 程序可能的运行结果

分析：按照 JDBC 连接数据库的步骤，首先加载数据库的 JDBC 驱动程序，再创建数据库连接，然后通过 PreparedStatement 语句根据用户输入的商品名称查询商品的状态，如果返回的结果集不为空，则显示相应商品的状态信息；反之，则显示“没有商品的相关信息”。最后关闭数据库连接。为了保证能正确关闭数据库连接，将 close()语句放在 finally 语句块中。

【例 14.3】 显示已有商品单价的 JSP 页面。

```
<%@ page language="java" contentType="text/html; charset=gb2312"
    pageEncoding="gb2312" import="java.sql.*"%>
<! DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<title>显示商品的单价</title>
</head>
<body>
```

```
商品单价列表:
<table border="1">
    <tr>
        <td>编号</td>
        <td>名称</td>
        <td>价格</td>
    </tr>
    <%
        String url="jdbc:derby:testDB;create=true";
        Connection con=null;
        Statement stmt=null;
        String query="SELECT * FROM commodity";

        try {
            //加载 JDBC 驱动程序
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            //建立连接
            con=DriverManager.getConnection(url);
            //创建 Statement 语句
            stmt=con.createStatement();
            //执行 SQL 语句
            ResultSet rs=stmt.executeQuery(query);

            while(rs.next()) {                    //通过循环语句检索结果
                int id=rs.getInt("id");
                String name=rs.getString("name");
                float price=rs.getFloat("price");
                out.println("<tr><td>"+id+"</td><td>"+name
                        +"</td><td>"+price+"</td></tr>");
            }
        } catch (ClassNotFoundException e) {
            out.print("类没有找到异常:");
            out.println(e.getMessage());
        } catch (SQLException e) {
            out.println("SQL 异常:"+e.getMessage());
        } finally {
            if (stmt !=null) {
                try {
                    stmt.close();                 //关闭 Statement 语句
                } catch (SQLException e) {
                }
                stmt=null;
            }
            if (con !=null) {
                try {
```

```
                    con.close();              //关闭连接
                } catch (SQLException e) {
                }
                con=null;
            }
        }
    %>
</table>
</body>
</html>
```

程序可能的运行结果如图 14-6 所示。



图 14-6　例 14.3 程序可能的运行结果

分析：在 JSP 中访问数据库的操作步骤与在 Java 应用程序中的操作步骤是完全一致的。为了使用 JDBC 中的相关类，在 JSP 的页面指令中使用了 import="java.sql.*"。通过 JSP 内置对象 out 将查询结果输出到网页中。

14.4.7　事务处理

数据库事务(Database Transaction)是数据库操作中的重要概念。它是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。例如，如果用户 A 要向用户 B 转账 1000 元，此时银行管理系统至少需要执行两条数据库操作语句，一条语句是对用户 A 的账户减少 1000 元，另一条语句是对用户 B 的账户增加 1000 元。这两条操作应该是捆绑在一起的，不能只执行其中一条语句，否则会引起数据不一致的问题。事务处理可以确保除非事务性单元内的所有操作都成功完成，否则不会永久更新面向数据的资源。数据库事务需要满足 4 个特性：原子性(Atomic)、一致性(Consistency)、隔离性(Isolation)和持久性(Durabiliy)，简称 ACID。

(1) 原子性：表示组成一个事务的多个数据库操作是一个不可分隔的原子单元，只有所有的操作执行成功，整个事务才提交，事务中任何一个数据库操作失败，已经执行的任何操作都必须撤销，让数据库返回到初始状态。

(2) 一致性：事务操作成功后，数据库所处的状态和它的业务规则是一致的，即数据不

会被破坏。

(3) 隔离性：在并发数据操作时，不同事务的操作之间不会相互产生干扰。用户可以通过指定连接的事务隔离级别来实现事务间的隔离性。ANSI/ISO SQL 92 标准定义了 4 个等级的事务隔离级别：READ UNCOMMITED、READ COMMITTED、REPEATABLE READ 和 SERIALIZABLE。按照上述 4 个等级的排列顺序，其保证数据一致性的能力越来越强，而数据并发处理的能力越来越弱。一般情况下，推荐使用 REPEATABLE READ 级别。在 JDBC 中对应有 4 个值来表示事务隔离级别：TRANSACTION_READ_UNCOMMITTED、TRANSACTION_READ_COMMITTED、TRANSACTION_REPEATABLE_READ 和 TRANSACTION_SERIALIZABLE。

(4) 持久性：一旦事务提交成功，事务中所有的数据操作都必须被持久化到数据库。

典型的 JDBC 事务数据操作的代码如下：

```
Connection conn;
try{
    //①获取数据连接
    conn=DriverManager.getConnection();
    //②关闭自动提交的机制
    conn.setAutoCommit(false);
    //③设置事务隔离级别
    //④执行数据库操作
    Statement stmt=conn.createStatement();
    stmt.executeUpdate("INSERT INTO commodity VALUES(1,'电脑 1',2000, '停产') ");
    //⑤提交事务
    conn.commit();
}catch(Exception e){
    …
    //⑥回滚事务
    conn.rollback();
}finally{
…
}
```

14.4.8 实用案例

【例 14.4】 事务操作示例。

运用事务操作，同时修改指定商品的状态和单价。

```
package code1404;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.sql.*;
```

```
public class ModifyCommodityStatusAndPrice {

    public static void main(String[] args) throws IOException {
        String url="jdbc:derby:testDB;create=true";
        Connection con=null;
        PreparedStatement pstm=null;

        boolean isExisted=false;

        System.out.println("请输入要修改的商品名称:");
        String inputName=(new BufferedReader(new InputStreamReader(System.
        in))).readLine();

        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            con=DriverManager.getConnection(url);
        } catch (ClassNotFoundException e) {
            System.err.print("类没有找到异常:");
            System.err.println(e.getMessage());
        } catch (SQLException e) {
            System.err.println("SQL 异常:"+e.getMessage());
        }

        try {
            pstm=con.prepareStatement("SELECT * FROM commodity where name=?");
            pstm.setString(1, inputName);
            ResultSet rs=pstm.executeQuery();

            if (rs.next()) {
                isExisted=true;
                System.out.println("商品修改前的信息是:");
                int id=rs.getInt("id");
                String name=rs.getString("name");
                float price=rs.getFloat("price");
                String status=rs.getString("status");
                System.out.println(id+"\t"+name+"\t"+price+"\t"+status);
            } else {
                System.out.println("输入的商品不存在。");
            }
        } catch (SQLException e) {
            System.err.println("SQL 异常:"+e.getMessage());
        } finally {
            if (pstm !=null) {
                try {
                    pstm.close();
```

```
                } catch (SQLException e) {
                }
                pstm=null;
            }
            System.err.flush();
        }

        if (isExisted) {
            System.out.println("请输入商品新的状态：");
            String inputStatus= (new BufferedReader(new InputStreamReader
            (System.in))).readLine();
            System.out.println("请输入商品新的价格：");
            Double inputPrice=Double.valueOf((new BufferedReader(new
            InputStreamReader(System.in))).readLine());

            try {
                con.setAutoCommit(false);
                con.setTransactionIsolation(Connection.TRANSACTION_
                REPEATABLE_READ);

                pstm=con.prepareStatement("UPDATE commodity SET price=? where
                name=?");
                pstm.setDouble(1, inputPrice.doubleValue());
                pstm.setString(2, inputName);
                pstm.executeUpdate();
                pstm.close();

                pstm=con.prepareStatement("UPDATE commodity SET status=? where
                name=?");
                pstm.setString(1, inputStatus);
                pstm.setString(2, inputName);
                pstm.executeUpdate();
                pstm.close();

                con.commit();
            } catch (SQLException e) {
                try {
                    con.rollback();
                } catch (SQLException e1) {
                    e1.printStackTrace();
                }
                System.err.println("SQL异常："+e.getMessage());
            } finally {
                if (pstm !=null) {
                    try {
```

```
                    pstm.close();
                } catch (SQLException e) {
                }
                pstm=null;
            }
            try {
                con.setAutoCommit(true);
            } catch (SQLException e) {
                e.printStackTrace();
            }
            System.err.flush();
        }

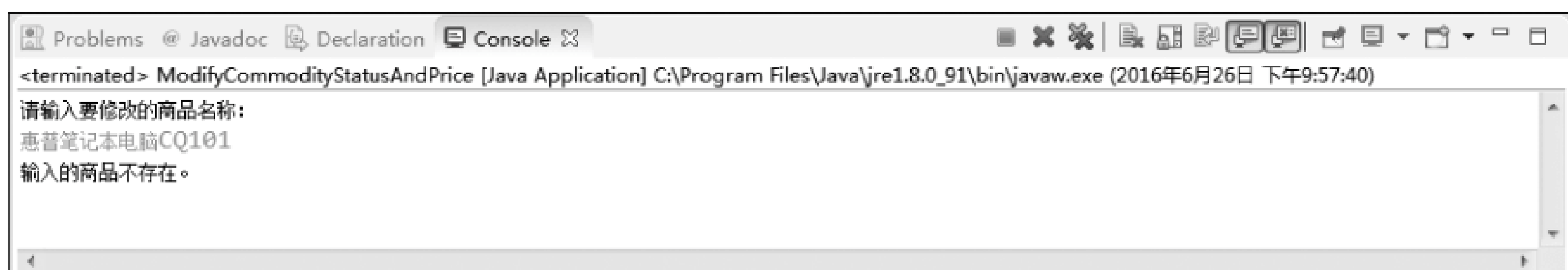
        try {
            pstm=con.prepareStatement("SELECT * FROM commodity where name=?");
            pstm.setString(1, inputName);
            ResultSet rs=pstm.executeQuery();

            if (rs.next()) {
                isExisted=true;
                System.out.println("商品修改后的信息是:");
                int id=rs.getInt("id");
                String name=rs.getString("name");
                float price=rs.getFloat("price");
                String status=rs.getString("status");
                System.out.println(id+"\t"+name+"\t"+price+"\t"+status);
            }
        } catch (SQLException e) {
            System.err.println("SQL 异常:"+e.getMessage());
        } finally {
            if (pstm !=null) {
                try {
                    pstm.close();
                } catch (SQLException e) {
                }
                pstm=null;
            }
            System.err.flush();
        }
    }

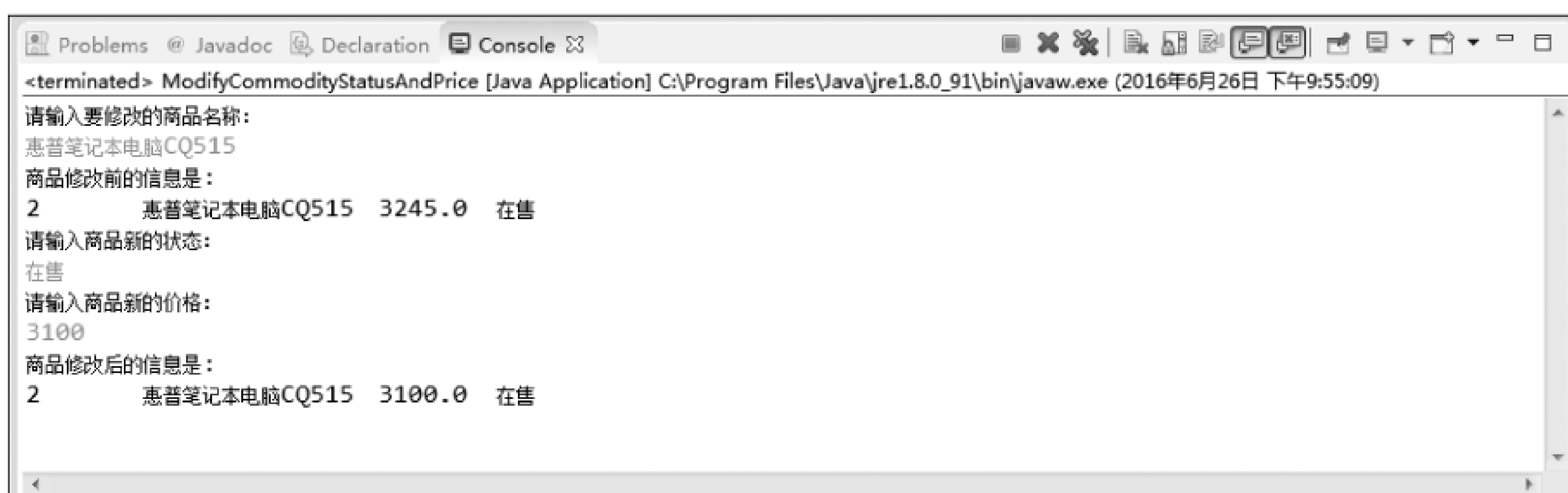
    if (con !=null) {
        try {
            con.close();
        } catch (SQLException e) {
```

```
            }
            con=null;
        }
    }
}
```

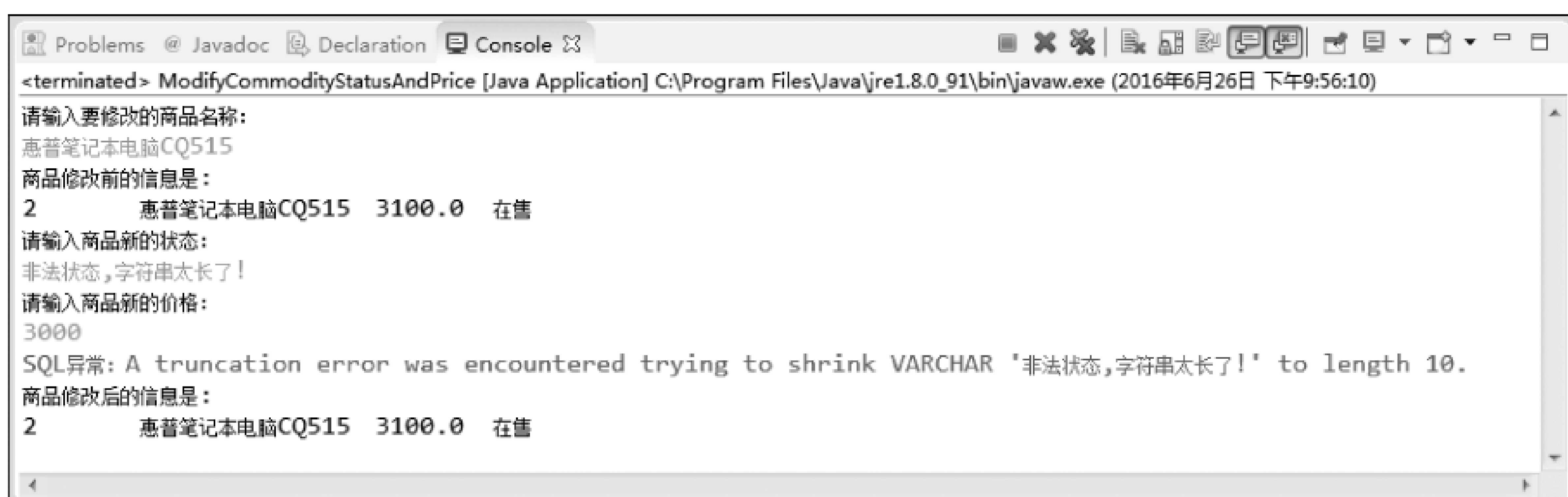
程序可能的运行结果如图 14-7 所示。


```
<terminated> ModifyCommodityStatusAndPrice [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (2016年6月26日 下午9:57:40)
请输入要修改的商品名称:
惠普笔记本电脑CQ101
输入的商品不存在。
```

(a)


```
<terminated> ModifyCommodityStatusAndPrice [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (2016年6月26日 下午9:55:09)
请输入要修改的商品名称:
惠普笔记本电脑CQ515
商品修改前的信息是:
2        惠普笔记本电脑CQ515   3245.0   在售
请输入商品新的状态:
在售
请输入商品新的价格:
3100
商品修改后的信息是:
2        惠普笔记本电脑CQ515   3100.0   在售
```

(b)


```
<terminated> ModifyCommodityStatusAndPrice [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (2016年6月26日 下午9:56:10)
请输入要修改的商品名称:
惠普笔记本电脑CQ515
商品修改前的信息是:
2        惠普笔记本电脑CQ515   3100.0   在售
请输入商品新的状态:
非法状态,字符串太长了!
请输入商品新的价格:
3000
SQL异常: A truncation error was encountered trying to shrink VARCHAR '非法状态,字符串太长了!' to length 10.
商品修改后的信息是:
2        惠普笔记本电脑CQ515   3100.0   在售
```

(c)

图 14-7　例 14.4 程序可能的运行结果

分析: 利用 JDBC 的事务处理操作,只有当商品价格修改操作和商品状态修改操作全部执行成功时,才会执行 commit 操作,将修改保存到数据库中;反之,只要其中任一操作失败,都会调用 rollback 操作,将数据库状态回滚到修改前的状态。

14.5　JDBC 实训任务

【任务描述】

通过程序完成以下操作:

(1) 在 testDB 数据库中创建商品表(commodity),该表有 4 个字段:商品编号(id)、商品名称(name)、单价(price)和商品状态(status)。

(2) 向商品表中添加新商品信息。

(3) 更改所有商品的单价,使之提高 10%。

（4）删除已经停产的商品信息。

其中第（2）～（4）步在执行相关操作后，都要显示操作之后商品表的数据记录情况。

【任务分析】

按照JDBC访问数据库的步骤来完成任务所要求的操作。任务中4个操作的程序代码的主要区别在于执行的SQL语句不同，其中创建表操作使用CREATE TABLE语句、添加新商品信息使用INSERT语句，更改商品单价使用UPDATE语句，删除商品使用DELETE语句，显示商品表信息用SELECT语句。

任务中的4个操作分别用4个类来完成，其中CreateCommodityTable类完成操作（1）；InsertNewCommodity类完成操作（2）；UpdateCommodityPrice类完成操作（3）；DeleteCommodity类完成操作（4）。数据库使用JavaDB。

【任务解决】

程序代码如下：

```
//CreateCommodityTable.java
import java.sql.* ;

public class CreateCommodityTable {

    public static void main(String[] args) {
        String url="jdbc:derby:testDB;create=true";
        Connection con=null;
        Statement stmt=null;
        String createString;
        createString="CREATE TABLE commodity "
                +"(id INT NOT NULL PRIMARY KEY, "+"name VARCHAR(50), "
                +"price FLOAT, "+"status VARCHAR(10))";

        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            con=DriverManager.getConnection(url);
            stmt=con.createStatement();
            stmt.executeUpdate(createString);

            System.out.println("创建 commodity表成功。");

        } catch (ClassNotFoundException e) {
            System.err.print("类没有找到异常：");
            System.err.println(e.getMessage());
        } catch (SQLException e) {
            System.err.println("SQL异常："+e.getMessage());
        } finally {
            if (stmt !=null) {
                try {
                    stmt.close();
```

```
                } catch (SQLException e) {
                }
                stmt=null;
            }
            if (con !=null) {
                try {
                    con.close();
                } catch (SQLException e) {
                }
                con=null;
            }
        }
    }
}

//InsertNewCommodity.java

import java.sql.*;
public class InsertNewCommodity {
    public static void main(String[] args) {
        String url="jdbc:derby:testDB;create=true";
        Connection con=null;
        Statement stmt=null;
        String query="SELECT * FROM commodity";

        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            con=DriverManager.getConnection(url);
            stmt=con.createStatement();

            stmt.executeUpdate("INSERT INTO commodity "
                    +"VALUES(1,'联想笔记本电脑 G450L-TTH',2699, '停产')");
            stmt.executeUpdate("INSERT INTO commodity "
                    +"VALUES(2,'惠普笔记本电脑 CQ515',2950, '在售')");
            stmt.executeUpdate("INSERT INTO commodity "
                    +"VALUES(3,'索尼笔记本电脑 VPCEA27EC',5259, '在售')");

            ResultSet rs=stmt.executeQuery(query);

            System.out.println("商品信息列表:");
            System.out.println("编号\t名称\t\t\t价格\t状态");
            while(rs.next()) {
                int id=rs.getInt("id");
                String name=rs.getString("name");
                float price=rs.getFloat("price");
```

```
                String status= rs.getString("status");
                System.out.println(id+ "\t"+ name+ "\t"+ price+ "\t"
                        + status);
            }
        } catch (ClassNotFoundException e) {
            System.err.print("类没有找到异常：");
            System.err.println(e.getMessage());
        } catch (SQLException e) {
            System.err.println("SQL 异常："+ e.getMessage());
        } finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                }
                stmt= null;
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                }
                con= null;
            }
        }
    }
}

//UpdateCommodityPrice.java
import java.sql.*;
public class UpdateCommodityPrice {
    public static void main(String[] args) {
        String url= "jdbc:derby:testDB;create= true";
        Connection con= null;
        Statement stmt= null;
        String query= "SELECT * FROM commodity";

        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
            con= DriverManager.getConnection(url);
            stmt= con.createStatement();

            stmt.executeUpdate("UPDATE commodity SET price= price * 1.1");

            ResultSet rs= stmt.executeQuery(query);
```

```
            System.out.println("价格增加 10%之后,商品信息列表:");
            System.out.println("编号\t名称\t\t\t价格\t状态");
            while(rs.next()) {
                int id= rs.getInt("id");
                String name= rs.getString("name");
                float price= rs.getFloat("price");
                String status= rs.getString("status");
                System.out.println(id+ "\t"+ name+ "\t"+ price+ "\t"
                        + status);
            }
        } catch (ClassNotFoundException e) {
            System.err.print("类没有找到异常:");
            System.err.println(e.getMessage());
        } catch (SQLException e) {
            System.err.println("SQL 异常:"+ e.getMessage());
        } finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                }
                stmt= null;
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                }
                con= null;
            }
        }
    }
}

//DeleteCommodity.java
import java.sql.* ;
public class DeleteCommodity {
    public static void main(String[] args) {
        String url= "jdbc:derby:testDB;create= true";
        Connection con= null;
        Statement stmt= null;
        String query= "SELECT *  FROM commodity";

        try {
            Class.forName("org.apache.derby.jdbc.EmbeddedDriver");
```

```
            con= DriverManager.getConnection(url);
            stmt= con.createStatement();

            stmt.executeUpdate("DELETE FROM commodity WHERE status= '停产'");

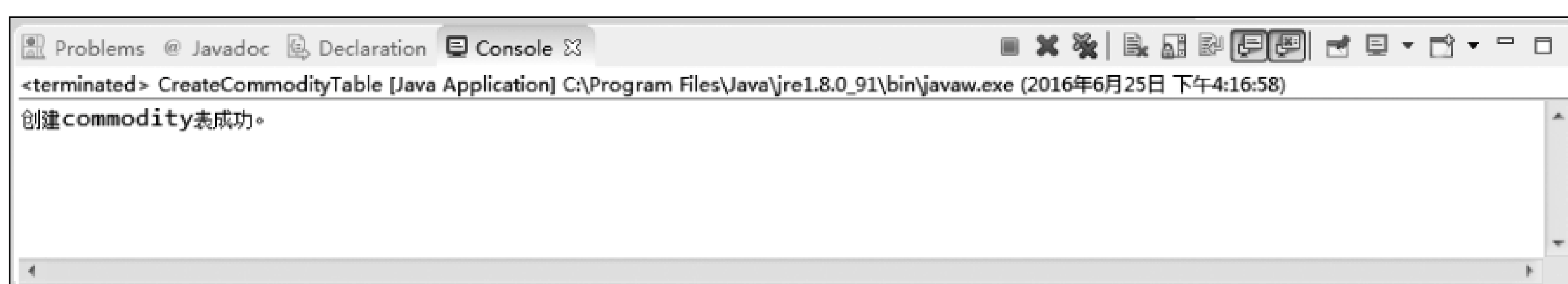
            ResultSet rs= stmt.executeQuery(query);

            System.out.println("删除停产商品之后的商品信息列表:");
            System.out.println("编号\t名称\t\t\t价格\t状态");
            while(rs.next()) {
                int id= rs.getInt("id");
                String name= rs.getString("name");
                float price= rs.getFloat("price");
                String status= rs.getString("status");
                System.out.println(id+ "\t"+ name+ "\t"+ price+ "\t"
                        + status);
            }
        } catch (ClassNotFoundException e) {
            System.err.print("类没有找到异常:");
            System.err.println(e.getMessage());
        } catch (SQLException e) {
            System.err.println("SQL 异常:"+ e.getMessage());
        } finally {
            if (stmt != null) {
                try {
                    stmt.close();
                } catch (SQLException e) {
                }
                stmt= null;
            }
            if (con != null) {
                try {
                    con.close();
                } catch (SQLException e) {
                }
                con= null;
            }
        }
    }
}
```

程序可能的运行结果如图 14-8 所示。

习题与思考

1. 常用的 SQL 语句有哪些?其作用是什么?



(a) 运行CreateCommodityTable.java的结果



(b) 运行InsertNewCommodity.java的结果



(c) 运行UpdateCommodityPrice.java的结果



(d) 运行DeleteCommodity.java的结果

图 14-8　JDBC 实训任务运行结果

2. JDBC 访问数据库的基本流程是什么?

3. Statement 对象和 PreparedStatement 对象的区别是什么?

第四篇　实 例 篇

第15章 Java应用开发案例

本章学习目标

俗话说“读万卷书,行万里路”。在前面的章节里已经详细地介绍了Java的各种知识,本章的两个应用开发案例——“基于Web的简单学生信息管理系统”和“推箱子游戏”将综合运用这些知识。“基于Web的简单学生信息管理系统”涉及数据库和Web相关的知识。“推箱子游戏”是基于Swing的GUI程序。希望通过这两个案例的学习,能够协助读者顺利地完成“行万里路”的过程。

15.1 基于Web的简单学生信息管理系统

15.1.1 程序的基本结构

本节将讲解一个基于Web能够对学生信息进行添加、修改、删除和查看的信息管理系统的实现过程。涉及的知识主要覆盖第12章～14章。系统选择用Eclipse IDE for Java EE Developers作为开发工具,Microsoft SQL Server 2000作为数据库。建议读者在阅读下面的内容之前,首先安装好开发工具与数据库。由于本书不是专门讲解数据库知识的书籍,因此开发案例尽量简化了数据库的相关操作。数据库只包含一个存储学生信息的表格,用于存储的学生信息包括学号(id)、姓名(name)、年龄(age)、性别(sex)、专业(major)、学院(college)和简介(introduction)。该表格的名称为student,对应的建表SQL语句为:

```
create table student(
id varchar(10) not null,
name varchar(50) not null,
age int not null,
sex int not null,
major varchar(50) not null,
college varchar(50) not null,
introduction varchar(500),primary key(id)
)
```

图 15-1 是学生信息管理系统的工程结构。其中,Java Resources:src 目录下放置的是 Java 源文件,包括 Student.java、StudentManager.java、StudentServlet.java,各 Java 文件的主要功能如下。

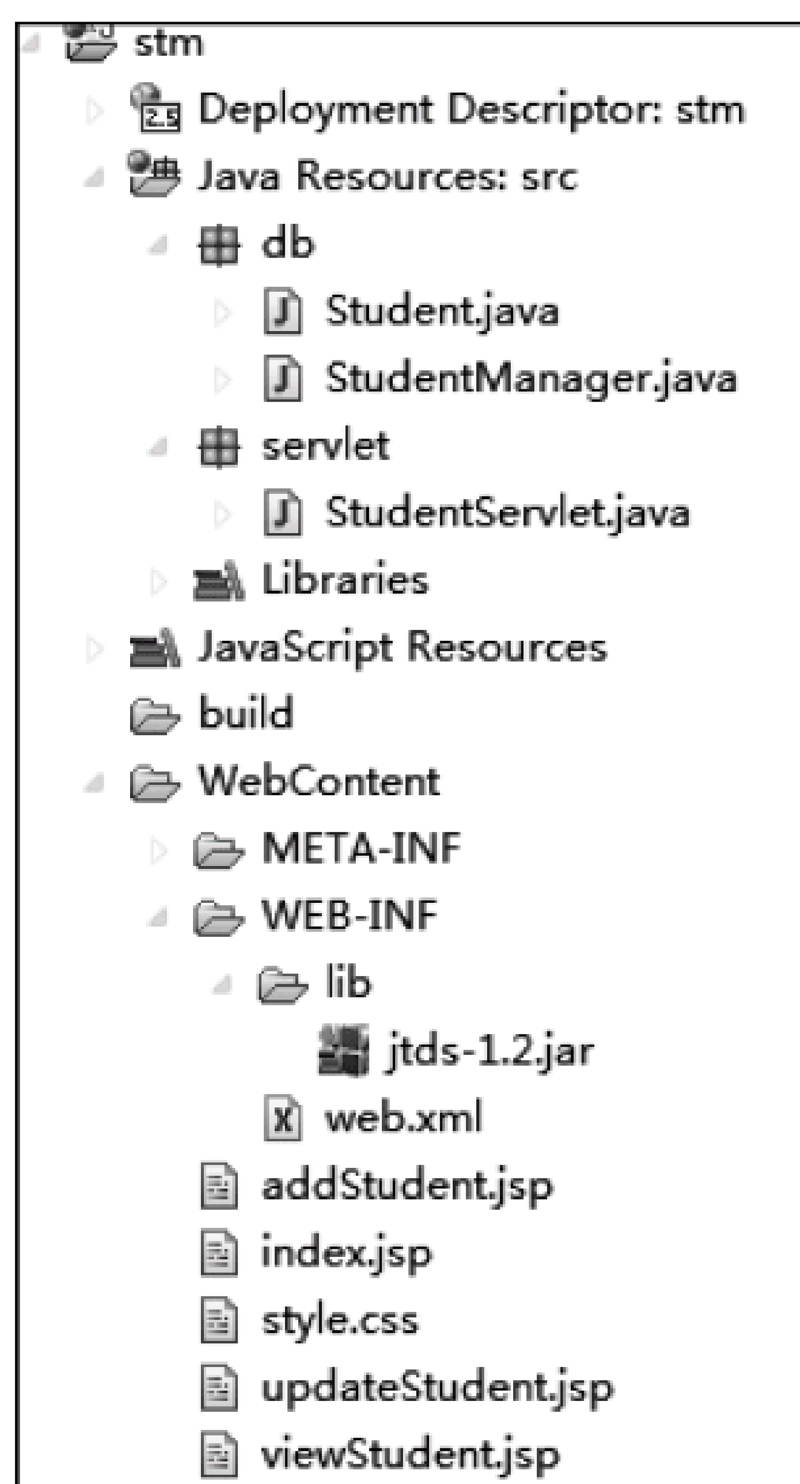


图 15-1　学生信息管理系统的工程结构

(1) Student.java:包含 Student 类,用于表示数据表格 student 中的一行记录。

(2) StudentManager.java:包含 StudentManager 类,用于对数据表格 student 中的记录进行操作,包括添加、修改、删除和查询等。

(3) StudentServlet.java:包含 StudentServlet 类,是一个 Servlet,映射到/deleteStudent、/addStudent 和/updateStudent 这 3 个 URL 地址,用于处理删除、添加和修改学生信息的 Web 请求。

WebContent 目录下包含 web.xml、jtds-1.2.jar、index.jsp、addStudent.jsp、updateStudent.jsp、viewStudent.jsp 和 style.css 等文件,其主要功能如下。

(1) web.xml:Web 工程的配置文件,配置了 Servlet 映射等信息。

(2) jtds-1.2.jar:SQL Server 2000 的数据库 JDBC 驱动文件。jtds 不是微软的官方 JDBC 驱动,但比官方驱动的性能更佳。它的下载地址为:http://sourceforge.net/projects/jtds。读者也可以将该驱动文件替换为微软的官方驱动。该文件必须放置在 WebContent/WEB-INF/lib 目录下,只有这样程序在运行时才能加载驱动。

(3) index.jsp:学生信息管理系统的首页面。

(4) addStudent.jsp:添加学生信息的页面。

(5) updateStudent.jsp:修改学生信息的页面。

(6) viewStudent.jsp:查看学生信息的页面。

(7) style.css:网页的样式单,能够让网页有更好的显示效果。

Web 工程各个文件之间的关系如图 15-2 所示。index.jsp、addStudent.jsp、updateStudent.jsp 和 viewStudent.jsp 是整个系统与用户交互的页面。在 index.jsp 页面通过超链接将 addStudent.jsp、updateStudent.jsp 和 viewStudent.jsp 联系起来。StudentServlet 负责处理删除、添加和修改学生信息的 Web 请求,它被映射为 3 个 URL 地址,即/deleteStudent、/addStudent 和/updateStudent,能够分别被 index.jsp、addStudent.jsp 和 updateStudent.jsp 调用。StudentManager 封装了全部对数据库的操作,它主要被 StudentServlet 所调用。index.jsp、updateStudent.jsp 和 viewStudent.jsp 涉及数据读取的部分也调用了 StudentManager。Student 用于表示 student 表中的一行记录,因此在程序中只要涉及 student 表中的数据的文件,例如 index.jsp、updateStudent.jsp、viewStudent.jsp 和 StudentServlet 等,都会使用 Student 类。

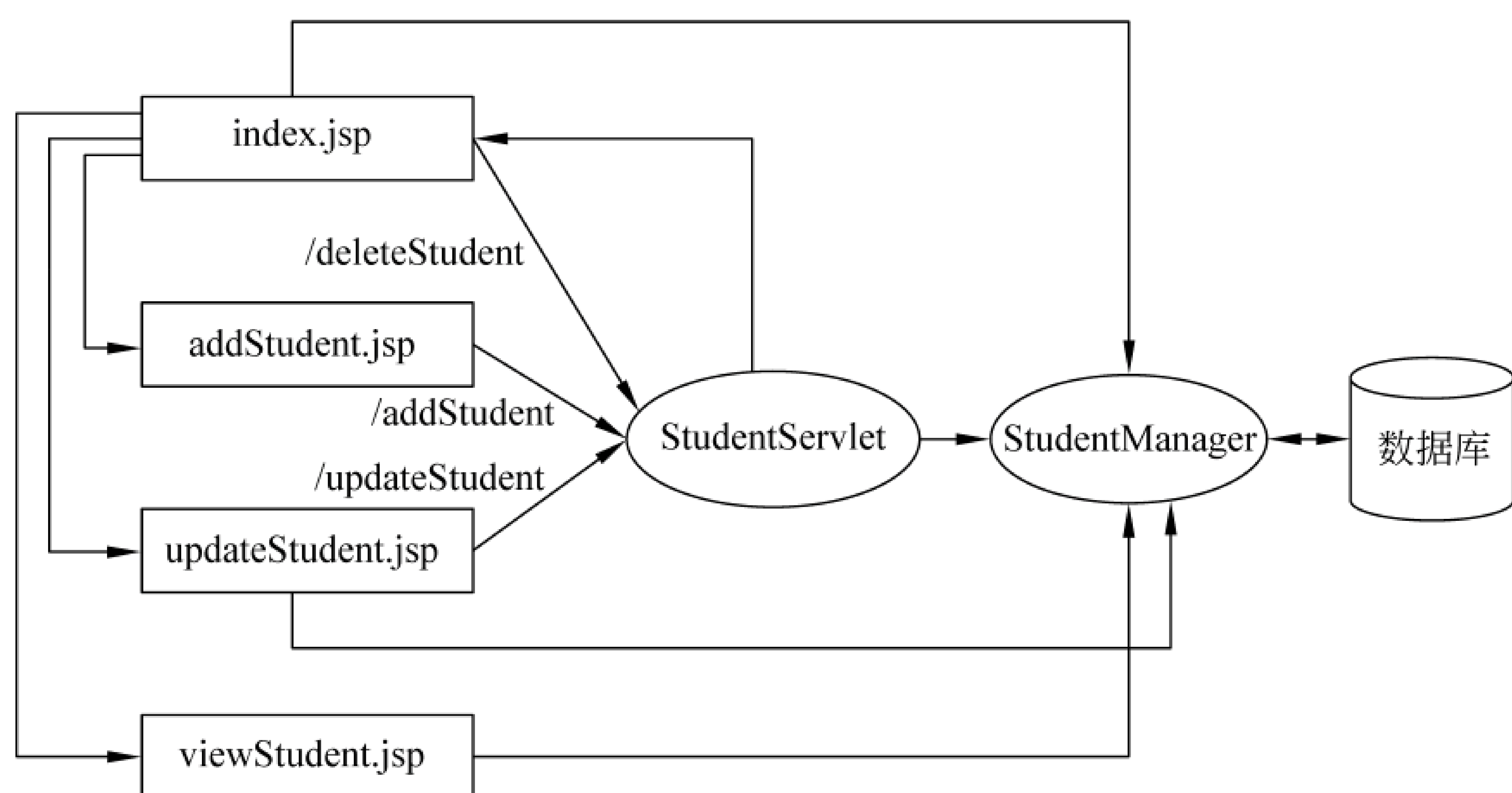


图 15-2　程序文件间的关系

15.1.2　数据库操作的实现

例 15.1 是 Student 类的代码。Student 类表示 student 表中的一行数据，它包含 id、name、age、sex、major、college 和 introduction 属性。这些属性与 student 表中的字段一一对应。由于属性的访问权限设置为 private，因此 Student 类还包含一系列的 get 和 set 函数用于提供对类属性的读取和写入操作。

【例 15.1】　Student. java。

```
package db;
public class Student {
    //学号
    private String id;
    //姓名
    private String name;
    //年龄
    private int age;
    //性别
    private int sex;
    //专业
    private String major;
    //学院
    private String college;
    //简介
    private String introduction;
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id= id;
    }
```

```
        public String getName() {
            return name;
        }
        public void setName(String name) {
            this.name=name;
        }
        public int getAge() {
            return age;
        }
        public void setAge(int age) {
            this.age=age;
        }
        public int getSex() {
            return sex;
        }
        public void setSex(int sex) {
            this.sex=sex;
        }
        public String getMajor() {
            return major;
        }
        public void setMajor(String major) {
            this.major=major;
        }
        public String getCollege() {
            return college;
        }
        public void setCollege(String college) {
            this.college=college;
        }
        public String getIntroduction() {
            return introduction;
        }
        public void setIntroduction(String introduction) {
            this.introduction=introduction;
        }
    }
```

StudentManager 类包含了全部的对 student 表的操作。例 15.2 给出了对应的代码。StudentManager 类包含一系列的静态方法,其作用如下。

(1) getConnection:用于建立与数据库的 JDBC 连接。本书所使用的数据库 url 地址、用户名、密码可能与读者安装的数据库有所差异,请读者注意修改对应的代码。由于使用了 jtds 数据库驱动,因此驱动的名称为 net.sourceforge.jtds.jdbc.Driver。读者也可以使用微软官方的 JDBC 驱动,但是需要将相应的驱动文件复制到 WebContent/WEB-INF/lib,并修改此处的驱动名称。

(2) addStudent：添加学生信息。首先通过 getConnection 方法获取数据库连接，然后创建 PreparedStatement 的实例，将参数与 SQL 语句进行绑定，最后执行 SQL 语句并关闭数据库连接。函数如果执行成功，则返回 true；如果执行失败(如产生异常)，则返回 false。

(3) updateStudent：修改学生信息。程序逻辑与 addStudent 基本相同，区别在于执行的 SQL 语句为 update 语句。

(4) deleteStudent：根据学号删除学生信息。程序逻辑与 addStudent 基本相同，区别在于执行的 SQL 语句为 delete 语句。

(5) getStudent：根据学号查询学生信息。首先通过 getConnection 方法获取数据库连接，然后创建 PreparedStatement 的实例，将参数与 SQL 语句进行绑定，执行 SQL 语句，最后将获得的数据构造一个 Student 的实例并关闭数据库连接。函数如果执行成功，则返回 Student 的实例；如果执行失败或者无法查询到学生信息，则返回 null。

(6) getAllStudents：查询全部学生的信息。程序流程基本与 getStudent 相同。由于数据有多条，因此程序构造了一个类型为 ArrayList<Student>的 students 实例，用 students 来存放返回的数据。如果学生数量为 0，则 students 的长度为 0。

【例 15.2】 StudentManager.java。

```
package db;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.util.ArrayList;
public class StudentManager {
    //获取数据库连接
    private static Connection getConnection() throws Exception {
        //jdbc url 地址
        String url= "jdbc:jtds:sqlserver://127.0.0.1:1433/pubs";
        //数据库用户名
        String user= "sa";
        //数据库密码
        String pwd= "";
        //数据库驱动名
        String driver= "net.sourceforge.jtds.jdbc.Driver";
        //加载数据库驱动
        Class.forName(driver);
        //获取数据库连接
        Connection conn= DriverManager.getConnection(url, user, pwd);
        return conn;
    }
    //添加学生信息
    public static boolean addStudent(String id, String name, int age, int sex,
            String major, String college, String introduction) {
        try {
```

```
            Connection conn= getConnection();
            PreparedStatement st= conn
                    .prepareStatement("insert into student values(?,?,?,?,?,?,?)");
            st.setString(1, id);
            st.setString(2, name);
            st.setInt(3, age);
            st.setInt(4, sex);
            st.setString(5, major);
            st.setString(6, college);
            st.setString(7, introduction);
            st.execute();
            conn.close();
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    //修改学生信息
    public static boolean updateStudent(String id, String name, int age,
            int sex, String major, String college, String introduction) {
        try {
            Connection conn= getConnection();
            PreparedStatement st= conn
                    .prepareStatement("update student set name= ?,age= ?,sex= ?,major= ?,college= ?,
                    introduction= ? where id= ?");
            st.setString(1, name);
            st.setInt(2, age);
            st.setInt(3, sex);
            st.setString(4, major);
            st.setString(5, college);
            st.setString(6, introduction);
            st.setString(7, id);
            st.execute();
            conn.close();
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    //删除学生信息
    public static boolean deleteStudent(String id) {
        try {
            Connection conn= getConnection();
```

```
            PreparedStatement st= conn
                    .prepareStatement("delete student where id= ?");
            st.setString(1, id);
            st.execute();
            conn.close();
            return true;
        } catch (Exception e) {
            return false;
        }
    }

    //按照学号查询学生信息
    public static Student getStudent(String id) {
        Student student= null;
        try {
            Connection conn= getConnection();
            PreparedStatement st= conn
                    .prepareStatement("select *  from student where id= ?");
            st.setString(1, id);
            st.execute();
            ResultSet rs= st.getResultSet();
            if (rs.next()) {
                student= new Student();
                student.setId(rs.getString("id"));
                student.setName(rs.getString("name"));
                student.setAge(rs.getInt("age"));
                student.setSex(rs.getInt("sex"));
                student.setMajor(rs.getString("major"));
                student.setCollege(rs.getString("college"));
                student.setIntroduction(rs.getString("introduction"));
            }
            conn.close();
        } catch (Exception e) {
        }
        return student;
    }

    //查询所有学生信息
    public static ArrayList<Student> getAllStudents(){
        ArrayList<Student> students= new ArrayList<Student> ();
        try {
            Connection conn= getConnection();
            PreparedStatement st= conn
                    .prepareStatement("select *  from student");
            st.execute();
```

```
                ResultSet rs= st.getResultSet();
                while(rs.next()) {
                    Student student= new Student();
                    student.setId(rs.getString("id"));
                    student.setName(rs.getString("name"));
                    student.setAge(rs.getInt("age"));
                    student.setSex(rs.getInt("sex"));
                    student.setMajor(rs.getString("major"));
                    student.setCollege(rs.getString("college"));
                    student.setIntroduction(rs.getString("introduction"));
                    students.add(student);
                }
                conn.close();
            } catch (Exception e) {
            }
            return students;
        }
    }
```

15.1.3 处理 Web 请求

例 15.3 是 StudentServlet 的代码。在新建 StudentServlet 时,它被映射到 3 个 URL 地址,如图 15-3 所示。

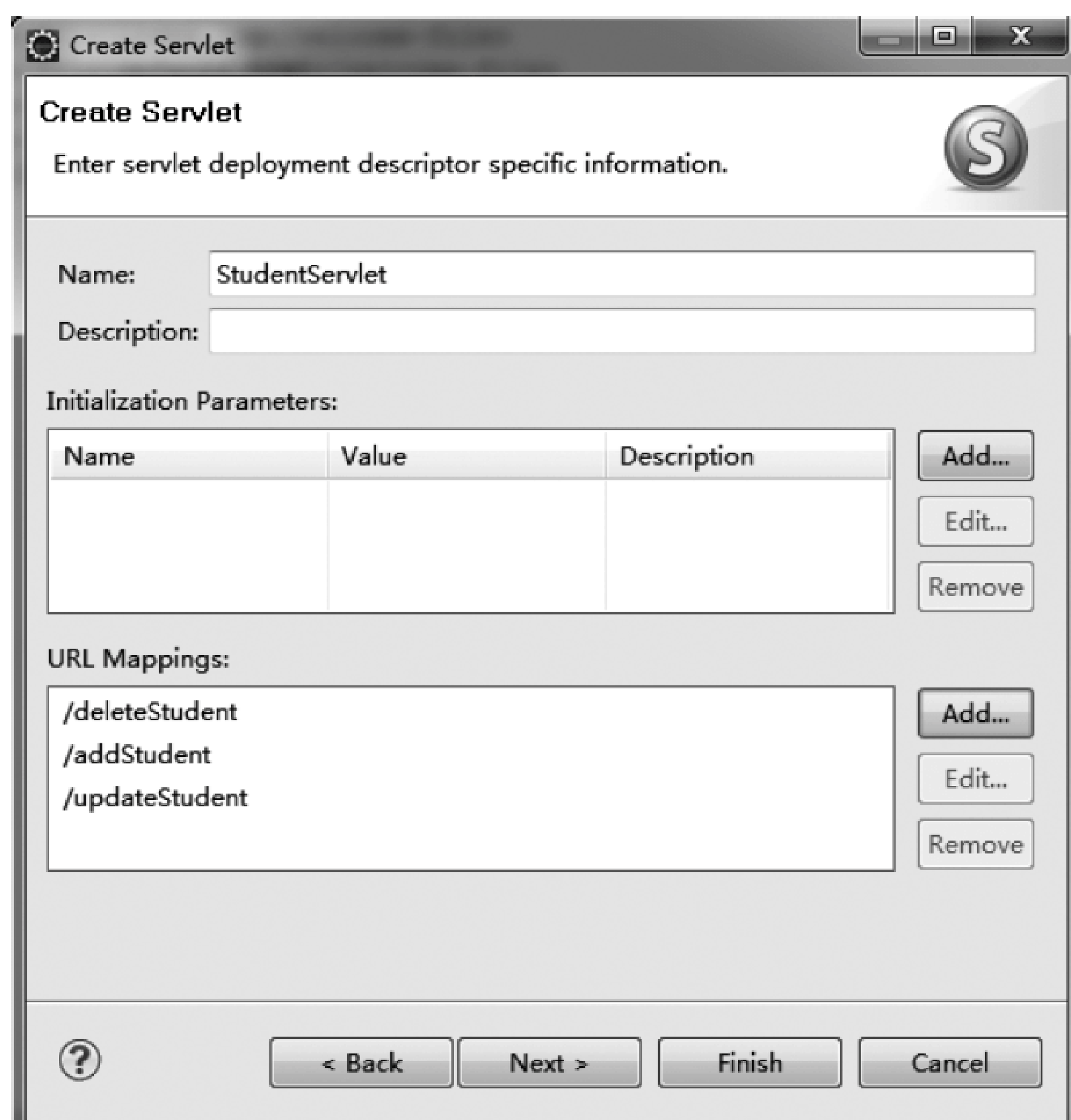


图 15-3 StudentServlet 的 URL 映射

StudentServlet 用 request.getRequestURI()方法来获取浏览器请求的服务器 URL 地址，然后根据 URL 地址来执行不同的程序逻辑。如果浏览器请求的 URL 地址结尾为 /deleteStudent，则执行 doDeleteStudent 方法。如果浏览器请求的 URL 地址结尾为 /addStudent，则执行 doAddStudent 方法。如果浏览器请求的 URL 地址结尾为 /updateStudent，则执行 doUpdateStudent 方法。这些方法的具体功能如下。

(1) doDeleteStudent：删除学生信息。通过 request.getParameter 方法获得学生的学号(id)，然后调用 StudentManager.deleteStudent 从数据库删除学生信息，最后调用 response.sendRedirect 方法跳转回到 index.jsp 页面。

(2) doAddStudent：添加学生信息。通过 request.getParameter 方法获得学生的学号、姓名和年龄等信息，然后调用 StudentManager.addStudent 方法将学生信息添加到数据库，最后调用 response.sendRedirect 方法跳转回到 index.jsp 页面。

(3) doUpdateStudent：修改学生信息。通过 request.getParameter 方法获得学生的学号、姓名和年龄等信息，然后调用 StudentManager.updateStudent 方法修改数据库中的学生信息，最后调用 response.sendRedirect 方法跳转回到 index.jsp 页面。

【例 15.3】 StudentServlet.java。

```
package servlet;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import db.StudentManager;
public class StudentServlet extends HttpServlet {
    //处理 GET 请求
    protected void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        //让 GET 和 POST 请求以相同的逻辑进行处理
        doPost(request, response);
    }

    protected void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        //设置编码为 GB18030,否则会出现中文乱码
        request.setCharacterEncoding("GB18030");
        if (request.getRequestURI().endsWith("/deleteStudent")) {
            //处理对 /deleteStudent 的请求
            doDeleteStudent(request, response);
        } else if (request.getRequestURI().endsWith("/addStudent")) {
            //处理对 /addStudent 的请求
            doAddStudent(request, response);
        } else if (request.getRequestURI().endsWith("/updateStudent")) {
            //处理对 /updateStudent 的请求
```

```
            doUpdateStudent(request, response);
        }
    }

    //删除学生信息
    private void doDeleteStudent(HttpServletRequest request,
            HttpServletResponse response) throws ServletException, IOException {
        String id= request.getParameter("id");
        StudentManager.deleteStudent(id);
        response.sendRedirect("index.jsp");
    }

    //添加学生信息
    private void doAddStudent(HttpServletRequest request,
            HttpServletResponse response) throws IOException{
        String id= request.getParameter("id");
        String name= request.getParameter("name");
        String age= request.getParameter("age");
        String sex= request.getParameter("sex");
        String major= request.getParameter("major");
        String college= request.getParameter("college");
        String introduction= request.getParameter("introduction");
        StudentManager.addStudent(id, name, new Integer(age), new Integer(sex),
                major, college, introduction);
        response.sendRedirect("index.jsp");

    }

    //修改学生信息
    private void doUpdateStudent(HttpServletRequest request,
            HttpServletResponse response) throws IOException {
        String id= request.getParameter("id");
        String name= request.getParameter("name");
        String age= request.getParameter("age");
        String sex= request.getParameter("sex");
        String major= request.getParameter("major");
        String college= request.getParameter("college");
        String introduction= request.getParameter("introduction");
        StudentManager.updateStudent(id, name, new Integer(age), new Integer(
                sex), major, college, introduction);
        response.sendRedirect("index.jsp");
    }
}
```

15.1.4 Web 页面

系统与用户交互的 Web 页面包括 index.jsp、viewStudent.jsp、addStudent.jsp 和

updateStudent.jsp。为了美化页面的显示效果,系统使用了 style.css 作为样式单。CSS 是 Web 页面设计常用的一类技术,但不是本书的重点,读者若想学习可以参考其他书籍。这里只给出其代码。

【例 15.4】 style.css。

```
TD.data {
    BACKGROUND-COLOR: #eeeee0;
    BORDER-BOTTOM: #cccc99 1px solid;
    BORDER-LEFT: #ffffff 1px solid;
    BORDER-RIGHT: #cccc99 1px solid;
    BORDER-TOP: #ffffff 1px solid;
    HEIGHT: 23px
}
TD.title {
    BACKGROUND-COLOR: #3988E4;
    BORDER-BOTTOM: #000033 1px solid;
    BORDER-LEFT: #669999 1px solid;
    BORDER-RIGHT: #000033 1px solid;
    BORDER-TOP: #669999 1px solid;
    COLOR: #ffffff;
    PADDING-BOTTOM: 0px;
    PADDING-LEFT: 10px;
    PADDING-RIGHT: 10px;
    PADDING-TOP: 0px
}
TD.header {
    BACKGROUND-COLOR: #ffffff;
    BORDER-BOTTOM: #cccccc 1px solid;
    BORDER-LEFT: #ffffff 1px solid;
    BORDER-RIGHT: #cccccc 1px solid;
    BORDER-TOP: #ffffff 1px solid;
    HEIGHT: 23px
}
TD {
    FONT-FAMILY: 宋体, Verdana, Helvetica, sans-serif;
    FONT-SIZE: 12px
}
body {
    background-color: #eeeeee;
    SCROLLBAR-FACE-COLOR: #dce0e2;
    SCROLLBAR-HIGHLIGHT-COLOR: #ffffff;
    SCROLLBAR-SHADOW-COLOR: #687888;
    SCROLLBAR-3DLIGHT-COLOR: #687888;
    SCROLLBAR-ARROW-COLOR: #6e7e88;
    SCROLLBAR-TRACK-COLOR: #bdbfc0;
```

```
        SCROLLBAR-DARKSHADOW-COLOR: #dce0e2;
        FONT-SIZE: 12px
}
A:link {
        COLOR: blue;
        TEXT-DECORATION: none
}

A:visited {
        COLOR: blue;
        TEXT-DECORATION: none
}

A:hover {
        COLOR: blue;
        TEXT-DECORATION: none
}
```

style.css 通过以下代码链接到 jsp 文件里,读者可以在后面 JSP 页面的代码中发现这条语句:

```
<link href="style.css" rel="stylesheet" type="text/css">
```

如果不使用样式单,Web 页面仍然可以显示,但显示效果会差一些。

index.jsp 是系统的首页,其显示效果如图 15-4 所示。页面上列出了全部的学生信息。“添加学生”链接到 addStudent.jsp,“查看”链接到 viewStudent.jsp,“修改”链接到 updateStudent.jsp,“删除”链接到/deleteStudent。当单击“删除”链接时会直接由 StudentServlet 删除对应的学生信息。单击其他链接则进入相应的页面。例 15.5 给出了 index.jsp 的代码。在页面上通过调用 StudentManager.getAllStudents 获得全部学生的信息,然后在 for 循环中将学生信息以表格的形式输出。

http://localhost:8080/stm/index.jsp

添加学生

全部学生信息								
学号	姓名	年龄	性别	专业	学院	操作		
0001	小王	20	男	计算机科学与技术	计算机学院	查看	修改	删除
0002	小红	19	男	信息安全	计算机学院	查看	修改	删除
0003	小张	20	男	网络工程	计算机学院	查看	修改	删除

图 15-4　index.jsp 的显示结果

【例 15.5】　index.jsp。

```
<%@ page language="java" pageEncoding="GB18030"%>
<%@ page import="db.StudentManager"%>
<%@ page import="db.Student"%>
<%@ page import="java.util.ArrayList"%>
```

```
<html>
<head>
<link href="style.css" rel="stylesheet" type="text/css">
<title>学生信息管理</title>
</head>
<body>
<div align="center">

<table width="600"  cellpadding="1" cellspacing="1">
<tr>
    <td colspan="9" align="right" height="30">
    <a href="addStudent.jsp">添加学生</a>
    </td>
</tr>
</table>
<table width="600"  cellpadding="1" cellspacing="1">
<tr>
    <td colspan="9" align="center" class="title" height="30">
    全部学生信息
    </td>
</tr>
<tr height="30">
    <td align="center" class="header">
    学号
    </td>
    <td align="center" class="header">
    姓名
    </td>
    <td align="center" class="header">
    年龄
    </td>
    <td align="center" class="header">
    性别
    </td>
    <td align="center" class="header">
    专业
    </td>
    <td align="center" class="header">
    学院
    </td>
    <td align="center" colspan="3" class="header">
    操作
    </td>
</tr>
<%
```

```
        ArrayList<Student> students=StudentManager.getAllStudents();
        for(Student student:students)
        {
    %>
    <tr height="30">
        <td align="center" class="data">
        <%=student.getId()%>
        </td>
        <td align="center" class="data">
        <%=student.getName()%>
        </td>
        <td align="center" class="data">
        <%=student.getAge()%>
        </td>
        <td align="center" class="data">
        <%=(student.getSex()==1?"男":"女")%>
        </td>
        <td align="center" class="data">
        <%=student.getMajor()%>
        </td>
        <td align="center" class="data">
        <%=student.getCollege()%>
        </td>
        <td align="center" class="data">
        <a href="viewStudent.jsp?id=<%=student.getId()%>">查看</a>
        </td>
        <td align="center" class="data">
        <a href="updateStudent.jsp?id=<%=student.getId()%>">修改</a>
        </td>
        <td align="center" class="data">
        <a href="deleteStudent?id=<%=student.getId()%>">删除</a>
    </td>
    </tr>
    <%
        }
    %>
    </table>
    </div>
    </body>
    </html>
```

addStudent.jsp 的显示结果如图 15-5 所示。用户可以在该页面上输入学生信息,然后单击“提交”按钮将数据提交到/addStudent,由 StudentServlet 进行处理。addStudent.jsp 的代码见例 15.6。



图 15-5　addStudent.jsp 的显示结果

【例 15.6】　addStudent.jsp。

```
<%@ page language="java" pageEncoding="GB18030"%>
<%@ page import="db.StudentManager"%>
<%@ page import="db.Student"%>
<%@ page import="java.util.ArrayList"%>
<html>
<head>
<link href="style.css" rel="stylesheet" type="text/css">
<title>添加学生信息</title>
</head>
<body>
<div align="center">
<form action="addStudent" method="post">
<table width="500" cellpadding="1" cellspacing="1">
    <tr>
        <td colspan="9" align="center" class="title" height="30">学生信息
</td>
    </tr>
    <tr height="30">
        <td align="center" class="header">学号:</td>
        <td align="center" class="data"><input name="id"></td>
    </tr>
    <tr height="30">
        <td align="center" class="header">姓名:</td>
        <td align="center" class="data"><input name="name"></td>
    </tr>
    <tr height="30">
```

```
            <td align="center" class="header">年龄:</td>
            <td align="center" class="data"><input name="age"></td>
        </tr>
        <tr height="30">
            <td align="center" class="header">性别:</td>
            <td align="center" class="data"><select name="sex">
                <option value="1">男</option>
                <option value="0">女</option>
            </select></td>
        </tr>
        <tr height="30">
            <td align="center" class="header">专业:</td>
            <td align="center" class="data"><input name="major"></td>
        </tr>
        <tr height="30">
            <td align="center" class="header">学院:</td>
            <td align="center" class="data"><input name="college"></td>
        </tr>
        <tr height="30">
            <td align="center" class="header">简介:</td>
            <td align="left" class="data"><textarea rows="10" cols="50"
                name="introduction"></textarea></td>
        </tr>
    </table>
    <input type="submit" value="提交"></form>
    </div>
    </body>
    </html>
```

viewStudent.jsp 的显示结果如图 15-6 所示。界面上显示了学生的详细信息。例 15.7 是 viewStudent.jsp 的代码。viewStudent.jsp 首先通过 request.getParameter 获得学生的学号(id),然后用 StudentManager.getStudent 得到该学生的信息,最后将该学生的信息以表格的形式输出。

http://localhost:8080/stm/viewStudent.jsp?id=0003

学生信息	
学号:	0003
姓名:	小张
年龄:	20
性别:	男
专业:	网络工程
学院:	计算机学院
简介:	小张的个人简介。。。

图 15-6　viewStudent.jsp 的显示结果

【例 15.7】 viewStudent.jsp。

```
<%@ page language="java" pageEncoding="GB18030"%>
<%@ page import="db.StudentManager"%>
<%@ page import="db.Student"%>
<%@ page import="java.util.ArrayList"%>
<html>
<head>
<link href="style.css" rel="stylesheet" type="text/css">
<title>查看学生信息</title>
</head>
<body>
<%
    String id=request.getParameter("id");
    Student student=StudentManager.getStudent(id);
%>
<div align="center">
<table width="500" cellpadding="1" cellspacing="1">
    <tr>
        <td colspan="9" align="center" class="title" height="30">学生信息
        </td>
    </tr>
    <tr height="30">
        <td align="center" class="header" width="100">学号:</td>
        <td align="center" class="data"><%=student.getId()%></td>
    </tr>
    <tr height="30">
        <td align="center" class="header" width="100">姓名:</td>
        <td align="center" class="data"><%=student.getName()%></td>
    </tr>
    <tr height="30">
        <td align="center" class="header" width="100">年龄:</td>
        <td align="center" class="data"><%=student.getAge()%></td>
    </tr>
    <tr height="30">
        <td align="center" class="header" width="100">性别:</td>
        <td align="center" class="data"><%=(student.getSex()==1 ? "男": "女")%>
        </td>
    </tr>
    <tr height="30">
        <td align="center" class="header" width="100">专业:</td>
        <td align="center" class="data"><%=student.getMajor()%></td>
    </tr>
    <tr height="30">
        <td align="center" class="header" width="100">学院:</td>
        <td align="center" class="data"><%=student.getCollege()%></td>
```

```
        </tr>
        <tr height="30">
            <td align="center" class="header" width="100">简介:</td>
            <td align="left" class="data"><%=student.getIntroduction()%></td>
        </tr>
    </table>
    </div>
    </body>
    </html>
```

updateStudent.jsp 的显示结果如图 15-7 所示。用户可以在该界面上修改学生信息,然后单击"提交"按钮,将数据提交给/updateStudent,由 StudentServlet 进行处理。例 15.8 是 updateStudent.jsp 的代码。updateStudent.jsp 首先通过 request.getParameter 获得学生的学号(id),然后用 StudentManager.getStudent 得到该学生的信息,最后将该学生的信息填写到表单元素中显示给用户。

图 15-7 updateStudent.jsp 的显示结果

【例 15.8】 updateStudent.jsp。

```
<%@ page language="java" pageEncoding="GB18030"%>
<%@ page import="db.StudentManager"%>
<%@ page import="db.Student"%>
<%@ page import="java.util.ArrayList"%>
<html>
<head>
<link href="style.css" rel="stylesheet" type="text/css">
<title>查看学生信息</title>
</head>
<body>
```

```
<%
    String id= request.getParameter("id");
    Student student= StudentManager.getStudent(id);
%>
<div align= "center">
<form action= "updateStudent" method= "post">
<table width= "500" cellpadding= "1" cellspacing= "1">
    <tr>
        <td colspan= "9" align= "center" class= "title" height= "30">学生信息
        </td>
    </tr>
    <tr height= "30">
        <td align= "center" class= "header">学号:</td>
        <td align= "center" class= "data"><input name= "id"
            value= "<%= student.getId()%>" readonly></td>
    </tr>
    <tr height= "30">
        <td align= "center" class= "header">姓名:</td>
        <td align= "center" class= "data"><input name= "name"
            value= "<%= student.getName()%>"></td>
    </tr>
    <tr height= "30">
        <td align= "center" class= "header">年龄:</td>
        <td align= "center" class= "data"><input name= "age"
            value= "<%= student.getAge()%>"></td>
    </tr>
    <tr height= "30">
        <td align= "center" class= "header">性别:</td>
        <td align= "center" class= "data"><select name= "sex">
            <%
                if (student.getSex()==1) {
                    out.println("<option value= \"1\" checked>男</option>");
                    out.println("<option value= \"0\">女</option>");
                } else {
                    out.println("<option value= \"1\">男</option>");
                    out.println("<option value= \"0\" checked>女</option>");
                }
            %>
        </select></td>
    </tr>
    <tr height= "30">
        <td align= "center" class= "header">专业:</td>
        <td align= "center" class= "data"><input name= "major"
            value= "<%= student.getMajor()%>"></td>
```

```
        </tr>
        <tr height="30">
            <td align="center" class="header">学院:</td>
            <td align="center" class="data"><input name="college"
                value="<%=student.getCollege()%>"></td>
        </tr>
        <tr height="30">
            <td align="center" class="header">简介:</td>
            <td align="left" class="data"><textarea rows="10" cols="50"
                name="introduction"><%=student.getIntroduction()%></textarea>
    </td>
        </tr>
    </table>
    <input type="submit" value="提交"></form>
    </div>
    </body>
    </html>
```

15.2 推箱子游戏

15.2.1 程序的基本结构

本节将讲解一个简单的推箱子游戏。推箱子是一种有趣的智力游戏,它的游戏规则是用角色将游戏地图上的所有箱子推到地图上特定的目标位置。推箱子游戏是一个桌面应用程序,其程序结构如图 15-8 所示。

src 目录下包含源程序文件,其主要功能如下。

(1) Game.java:包含 Game 类。该类是对游戏逻辑的封装。

(2) GameCanvas.java:包含 GameCanvas 类。该类是 JPanel 的子类,负责绘制游戏界面。

(3) GameFrame.java:包含 GameFrame 类。该类是游戏的窗体。

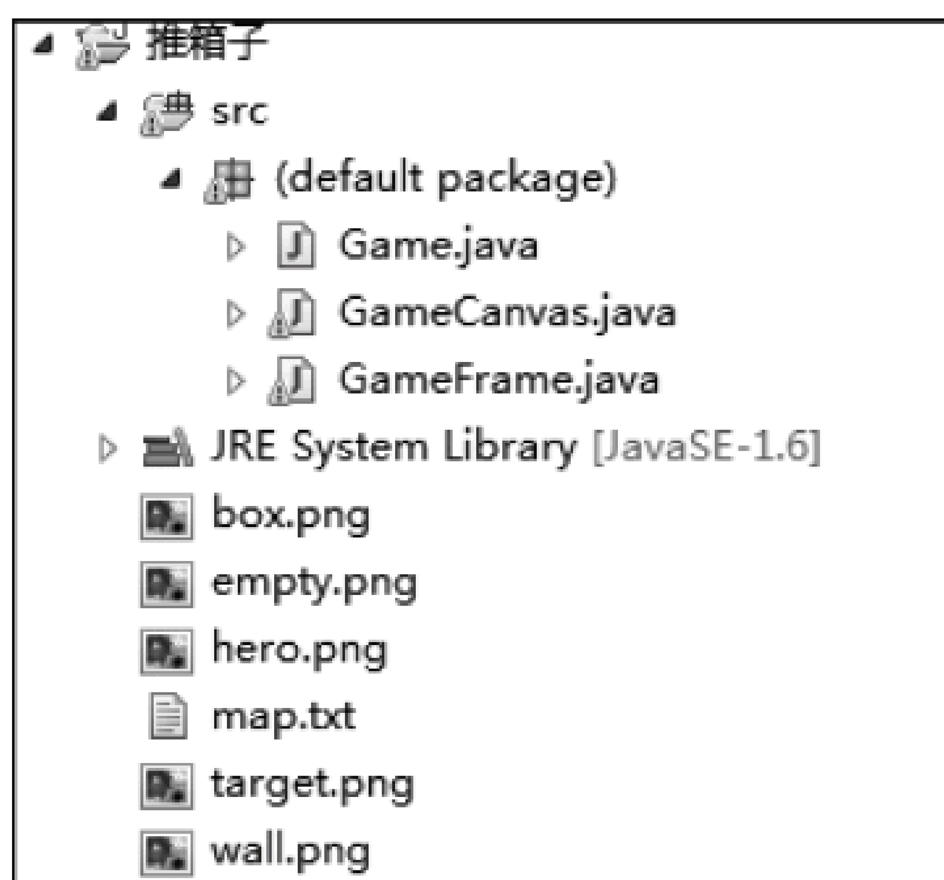


图 15-8 推箱子游戏的程序结构

box.png、empty.png、hero.png、target.png、wall.png 是游戏中所使用的图片资源,分别对应箱子、空地、游戏角色、推箱子的目标位置、墙。map.txt 文件存储了游戏地图。

15.2.2 游戏地图设计

程序用字符串二维数组来表示游戏地图。例如图 15-9 所示的游戏地图,可以表示为如下的二维数组:

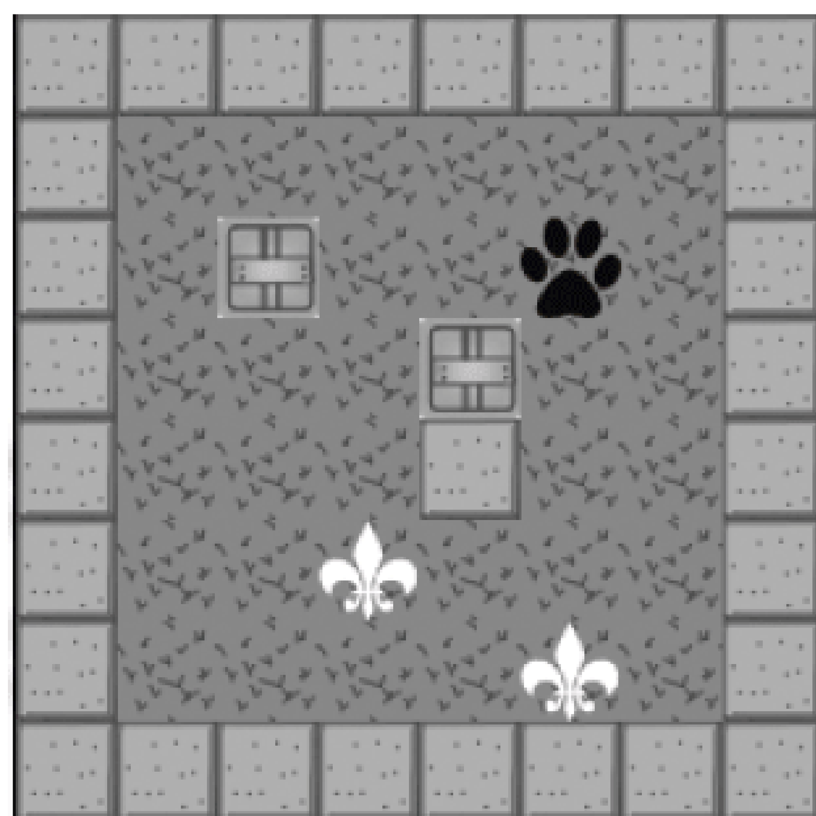


图 15-9　游戏地图

W W W W W W W W
W E E E E E E W
W E B E E H E W
W E E E B E E W
W E E E W E E W
W E E T E E E W
W E E E E T E W
W W W W W W W W

其中,W 表示墙,对应图片 wall.png;E 表示空地,对应图片 empty.png;B 表示箱子,对应图片 box.png;H 表示游戏角色,对应图片 hero.png;T 表示推箱子的目标位置,对应图片 target.png。数组中字母与图片的对应关系如图 15-10 所示。用于表示地图的二维数组保存在 map.txt 文件中,并由 Game 类在构造函数中加载。GameCanvas 类通过读取 Game 类的数据来进行游戏界面的绘制。

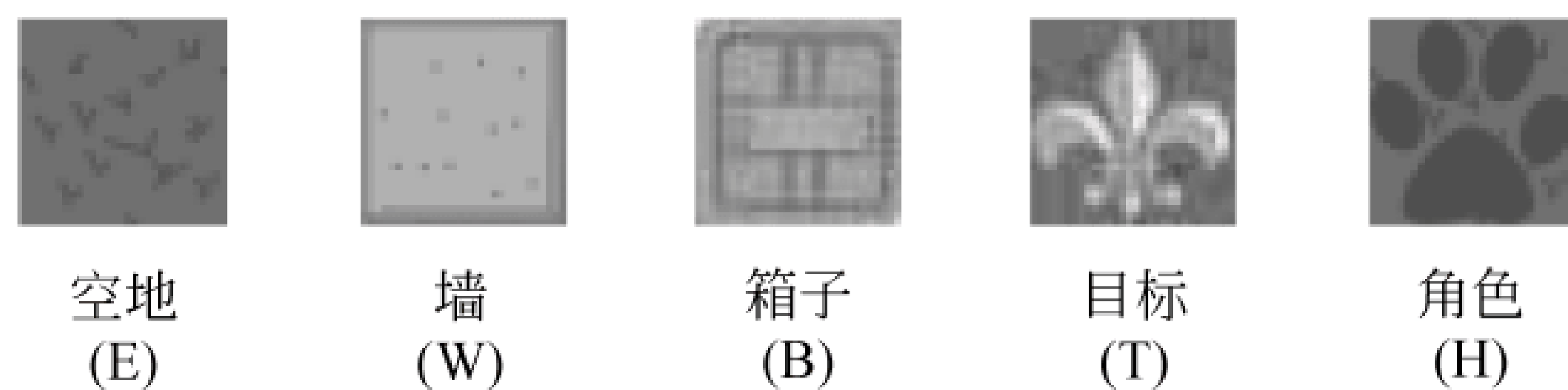


图 15-10　字母与图片的对应关系

15.2.3　程序逻辑

Game 类封装了全部的游戏逻辑,其程序代码如下。

【例 15.9】　Game.java。

```
import java.io.BufferedReader;
import java.io.FileReader;
public class Game {
    //游戏角色的坐标位置
    private int x;
    private int y;
    //游戏为 8×8的地图
    private String map[][]=new String[8][8];
    //记录推箱子的目标位置
    private String target[][]=new String[8][8];
    public Game() {
        //从 map 文件加载地图信息
        try {
            BufferedReader reader=new BufferedReader(new FileReader("map.txt"));
            //读取文件的一行
            String line=reader.readLine();
            int j=0;
```

```
            while(line != null) {
                //以空格为分隔符号,分割一行数据
                String strs[]= line.split(" ");
                for (int i= 0; i< 8; i+ + ) {
                    map[i][j]= strs[i];
                    if (strs[i].equals("T")) {
                        //记录目标位置
                        target[i][j]= "T";
                    } else {
                        //非目标位置记为 O
                        target[i][j]= "O";
                    }
                    //记录角色的初始位置
                    if (strs[i].equals("H")) {
                        this.x= i;
                        this.y= j;
                    }
                }
                line= reader.readLine();
                j+ + ;
            }
            reader.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    //角色向左移动
    public void moveLeft() {
        int a= x;
        int b= y;
        if(!map[a- 1][b].equals("W")) {
            if (map[a- 1][b].equals("B")) {
                if(!map[a- 2][b].equals("B") && ! map[a- 2][b].equals("W")) {
                    if (target[a][b].equals("T")) {
                        map[a- 1][b]= "H";
                        map[a][b]= "T";
                        map[a- 2][b]= "B";
                        x= a- 1;
                    } else {
                        map[a- 1][b]= "H";
                        map[a][b]= "E";
                        map[a- 2][b]= "B";
                        x= a- 1;
                    }
                }
```

```
            } else {
                if (target[a][b].equals("T")) {
                    map[a-1][b]="H";
                    map[a][b]="T";
                    x=a-1;
                } else {
                    map[a-1][b]="H";
                    map[a][b]="E";
                    x=a-1;
                }
            }
        }
    }
    //角色向右移动
    public void moveRight() {
        int a=x;
        int b=y;
        if(!map[a+1][b].equals("W")) {
            if (map[a+1][b].equals("B")) {
                if(!map[a+2][b].equals("B") && ! map[a+2][b].equals("W")) {
                    if (target[a][b].equals("T")) {
                        map[a+1][b]="H";
                        map[a][b]="T";
                        map[a+2][b]="B";
                        x=a+1;
                    } else {
                        map[a+1][b]="H";
                        map[a][b]="E";
                        map[a+2][b]="B";
                        x=a+1;
                    }
                }
            } else {
                if (target[a][b].equals("T")) {
                    map[a+1][b]="H";
                    map[a][b]="T";
                    x=a+1;
                } else {
                    map[a+1][b]="H";
                    map[a][b]="E";
                    x=a+1;
                }
            }
        }
    }
```

```
        //角色向上移动
        public void moveUp() {
            int a=x;
            int b=y;
            if(!map[a][b-1].equals("W")) {
                if (map[a][b-1].equals("B")) {
                    if(!map[a][b-2].equals("B") && ! map[a][b-2].equals("W")) {
                        if (target[a][b].equals("T")) {
                            map[a][b-1]="H";
                            map[a][b]="T";
                            map[a][b-2]="B";
                            y=b-1;
                        } else {
                            map[a][b-1]="H";
                            map[a][b]="E";
                            map[a][b-2]="B";
                            y=b-1;
                        }
                    }
                } else {
                    if (target[a][b].equals("T")) {
                        map[a][b-1]="H";
                        map[a][b]="T";
                        y=b-1;
                    } else {
                        map[a][b-1]="H";
                        map[a][b]="E";
                        y=b-1;
                    }
                }
            }
        }
        //角色向下移动
        public void moveDown() {
            int a=x;
            int b=y;
            if(!map[a][b+1].equals("W")) {
                if (map[a][b+1].equals("B")) {
                    if(!map[a][b+2].equals("B") && ! map[a][b+2].equals("W")) {
                        if (target[a][b].equals("T")) {
                            map[a][b+1]="H";
                            map[a][b]="T";
                            map[a][b+2]="B";
                            y=b-1;
                        } else {
```

```
                    map[a][b+1]="H";
                    map[a][b]="E";
                    map[a][b+2]="B";
                    y=b+1;
                }
            }
        } else {
            if (target[a][b].equals("T")) {
                map[a][b+1]="H";
                map[a][b]="T";
                y=b+1;
            } else {
                map[a][b+1]="H";
                map[a][b]="E";
                y=b+1;
            }
        }
    }
}
//判断游戏是否结束
public boolean isEnd() {
    for (int i=0; i<8; i++)
        for (int j=0; j<8; j++) {
            if (map[i][j].equals("T"))
                return false;
        }
    return true;
}
public String[][] getMap() {
    return map;
}
}
```

Game 类的成员变量 x 和 y 用于记录游戏角色的当前位置，map 数组记录游戏地图，target 数组用于记录推箱子的目标位置。这些成员变量在 Game 类的构造函数中依据 map.txt 文件的内容进行初始化。Game 类的构造函数首先创建一个针对 map.txt 文件的 BufferedReader。然后一行行地读取文件的内容。用 String 类的 split 方法将读取的每行字符串以“空格”为分隔符分割为单个字符。最后将分割后的内容保存到 map 数组。如果该字符表示游戏角色，则将 x 和 y 设置为该字符的位置，用于表示游戏角色的初始位置。如果该字符表示推箱子的目标位置，则设置 target 数组的对应位置为 T，反之设置为 O。

moveLeft、moveRight、moveUp 和 moveDown 方法用于控制游戏角色向左、向右、向上和向下移动。下面以游戏角色向左移动(moveLeft)为例，说明游戏角色的移动流程。moveRight、moveUp、moveDown 与 moveLeft 的程序逻辑类似，只是方向不同。如图 15-11 所示，初始状态时游戏角色位于坐标(a,b)处，向左移动的步骤如下：

(1) 判断(a－1,b)处(即角色左边 1 格)是否为墙。如果是,则不能移动;反之,则继续。

(2) 判断(a－1,b)处是否为箱子。如果是箱子,则需要继续判断箱子是否可以移动,进入步骤(4);如果不是箱子,则只需要移动角色,进入步骤(3)。

(3) 判断(a,b)处(游戏角色当前位置)是否为推箱子的目标位置。如果是,则设置 x＝a－1,y＝b,map[a－1][b]＝H,map[a][b]＝T;反之,设置 x＝a－1,y＝b,map[a－1][b]＝H,map[a][b]＝E。移动结束。

(4) 判断(a－2,b)处(游戏角色向左 2 格)是否为墙或者箱子。如果是,则不能移动,反之,则继续。

(5) 判断(a,b)处(游戏角色当前位置)是否为推箱子的目标位置。如果是,则设置 x＝a－1,y＝b,map[a－1][b]＝H,map[a][b]＝T,map[a－2][b]＝B;反之,设置 x＝a－1,y＝b,map[a－1][b]＝H,map[a][b]＝E,map[a－2][b]＝B。移动结束。

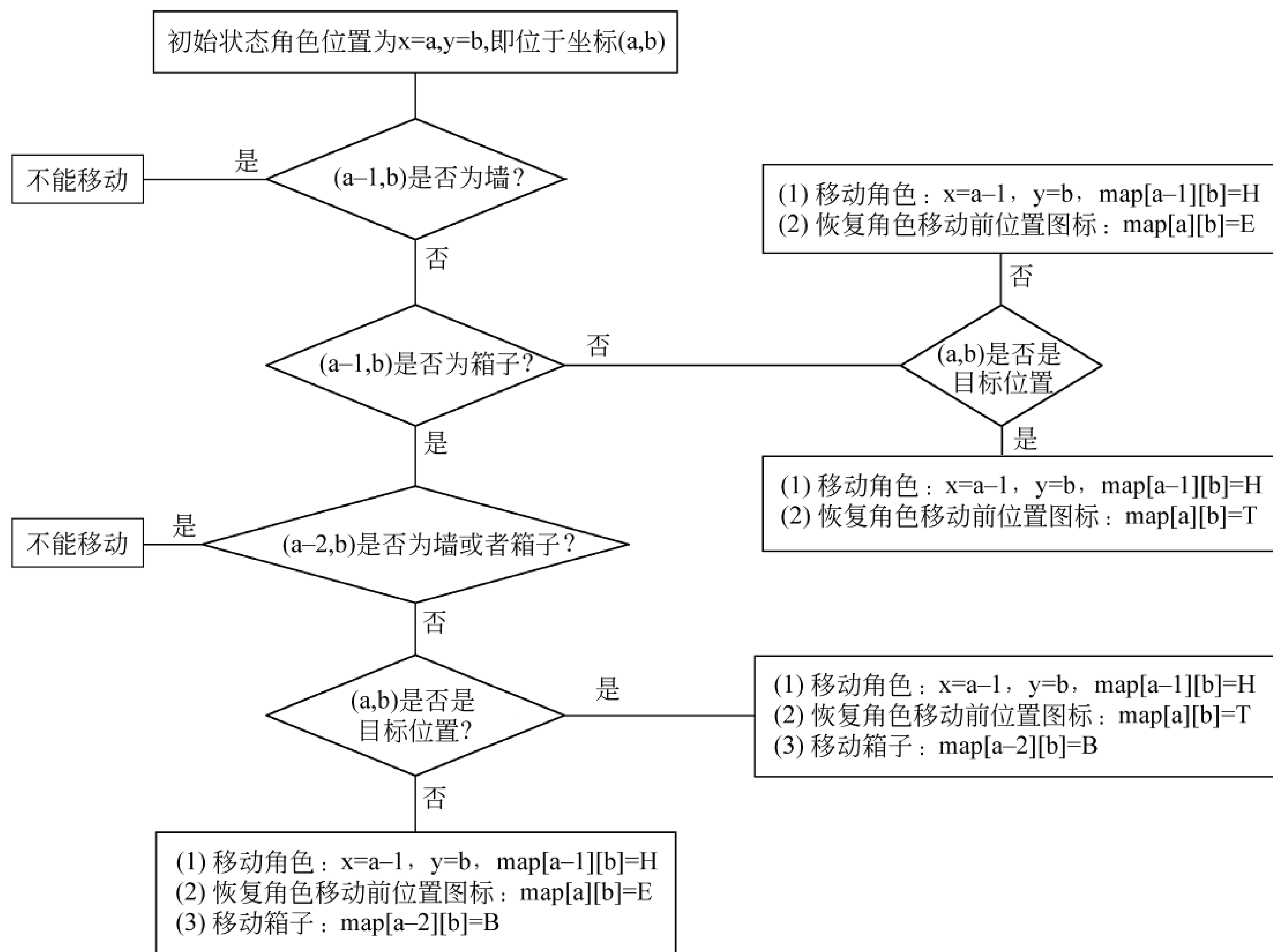


图 15-11　游戏角色向左移动

isEnd 方法用于判断游戏是否结束。推箱子游戏结束的条件为所有推箱子的目标位置上都有箱子存在,即 map 数组中没有任何元素的值为 T。

15.2.4　绘制游戏界面

GameCanvas 类是 JPanel 的子类,用于绘制游戏界面,其代码如下。

【例 15.10】　GameCanvas.java。

```
import java.awt.Graphics;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import javax.swing.JPanel;
public class GameCanvas extends JPanel {
    //箱子图片
    private BufferedImage box;
    //空地图片
    private BufferedImage empty;
    //角色图片
    private BufferedImage hero;
    //目标图片
    private BufferedImage target;
    //墙图片
    private BufferedImage wall;

    private Game game;
    public GameCanvas(Game game)
    {
        this.game=game;
        try {
            box=ImageIO.read(new File("box.png"));
            empty=ImageIO.read(new File("empty.png"));
            hero=ImageIO.read(new File("hero.png"));
            target=ImageIO.read(new File("target.png"));
            wall=ImageIO.read(new File("wall.png"));
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    //绘制游戏界面
    public void paint(Graphics g) {
        super.paint(g);
        BufferedImage image=null;
        for(int j=0;j<8;j++)
            for(int i=0;i<8;i++)
            {
                if(game.getMap()[i][j].equals("B"))
                {
                    image=box;
                }else if(game.getMap()[i][j].equals("E"))
                {
                    image=empty;
                }else if(game.getMap()[i][j].equals("H"))
                {
```

```
                        image=hero;
                    }else if(game.getMap()[i][j].equals("T"))
                    {
                        image=target;
                    }else if(game.getMap()[i][j].equals("W"))
                    {
                        image=wall;
                    }
                    g.drawImage(image, 0+i * image.getWidth(), 0+j * image.getHeight(),null);
                }
            }
        }
```

GameCanvas 类在构造函数中利用 ImageIO 类加载图片资源。然后在 paint (Graphics g)方法中根据 Game 类的 map 成员变量绘制游戏界面。Graphics 类实现了对图形上下文(graphics context)的封装,具有常用的图形绘制方法(如表 15-1 所示)。推箱子游戏使用 drawImage 方法实现了游戏界面的绘制。

表 15-1　Graphics 的常用的图形绘制方法

方　法　名	作　　用
drawLine	绘制直线
drawOval fillOval	绘制椭圆 填充椭圆
drawPolygon fillPolygon	绘制多边形 填充多边形
drawRect fillRect	绘制矩形 填充矩形
drawRoundRect fillRoundRect	绘制圆角矩形 填充圆角矩形
drawString	绘制字符串
draw3DRect fill3DRect	绘制带 3D 效果的矩形 填充带 3D 效果的矩形
drawArc fillArc	绘制弧形 填充弧形
drawImage	绘制图片,要使用 java. awt. Image 类及其子类
setColor	设置画笔颜色,要使用 java. awt. Color 类
setFont	设置字体,要使用 java. awt. Font 类

15.2.5　游戏窗体

GameFrame 类是游戏窗体。在窗体上包含了 GameCanvas 类的实例用于绘制游戏界面。GameFrame 类加入了按钮事件处理。当按方向键“上”时调用 Game 类的 moveUp 方法,按方向键“下”时调用 Game 类的 moveDown 方法,按方向键“左”时调用 Game 类的 moveLeft 方法,按方向键“右”时调用 Game 类的 moveRight 方法。当角色移动之后,调用

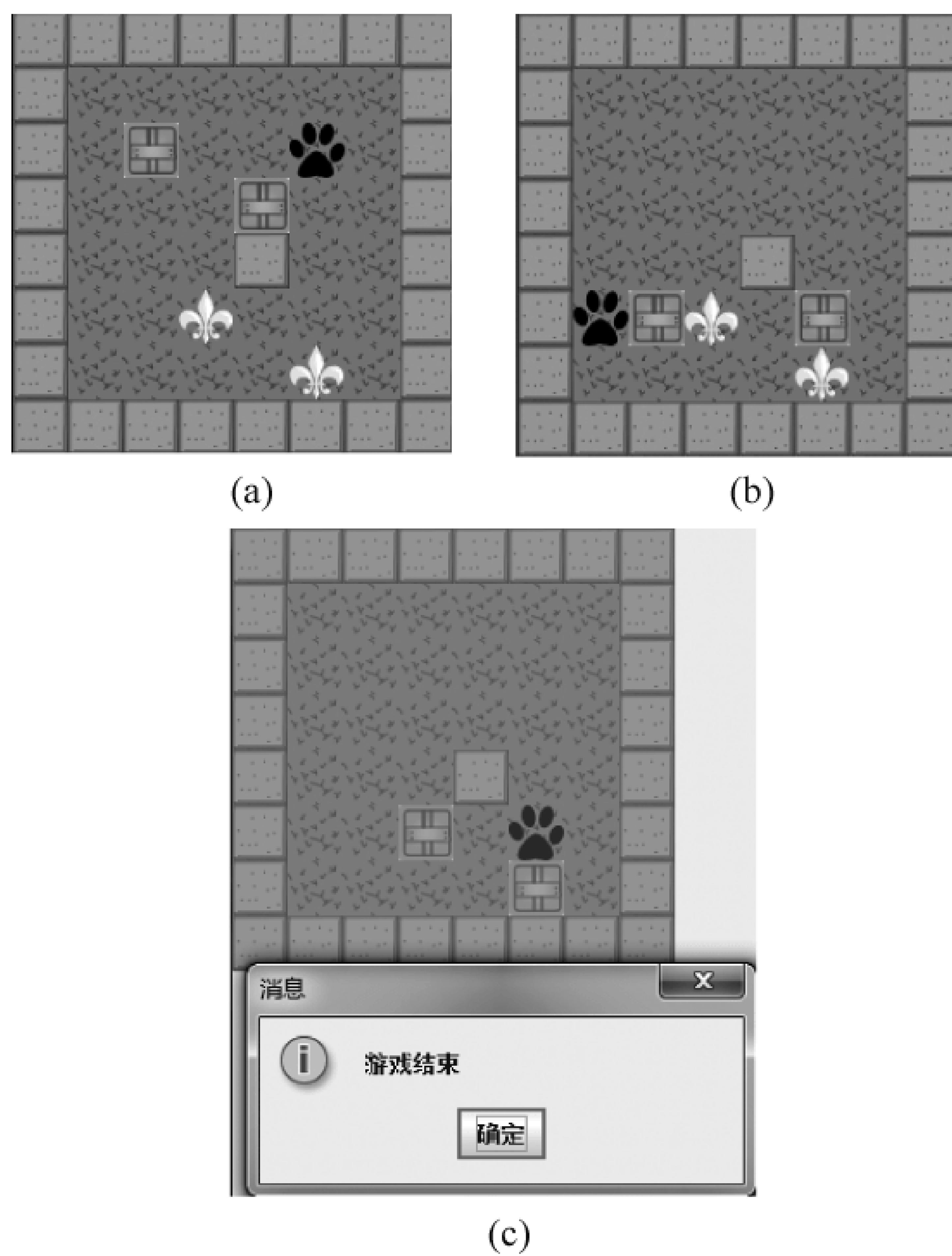
GameCanvas 的 repaint 方法实现游戏界面的重绘。最后调用 Game 类的 isEnd 方法判断游戏是否结束。GameFrame 类的代码如下。

【例 15.11】 GameFrame.java。

```
import java.awt.BorderLayout;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
public class GameFrame extends JFrame {
    private Game game;
    private GameCanvas canvas;
    public GameFrame()
    {
        game= new Game();
        canvas= new GameCanvas(game);
        this.setLayout(new BorderLayout());
        this.add(canvas,BorderLayout.CENTER);
        this.setSize(500,500);
        this.setLocationRelativeTo(null);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.addKeyListener(new KeyListener(){
            public void keyPressed(KeyEvent event) {
                if(event.getKeyCode()==KeyEvent.VK_UP)
                {
                    game.moveUp();
                }else if(event.getKeyCode()==KeyEvent.VK_DOWN)
                {
                    game.moveDown();
                }else if(event.getKeyCode()==KeyEvent.VK_LEFT)
                {
                    game.moveLeft();
                }else if(event.getKeyCode()==KeyEvent.VK_RIGHT)
                {
                    game.moveRight();
                }
                canvas.repaint();
                if(game.isEnd())
                    JOptionPane.showMessageDialog(null, "游戏结束");
            }
            public void keyReleased(KeyEvent event) {
            }
            public void keyTyped(KeyEvent event) {
            }
        });
```

```
        }
        public static void main(String[] args) {
            GameFrame frame= new GameFrame();
            frame.setVisible(true);
        }
}
```

推箱子游戏的程序运行效果如图 15-12 所示。



(a) (b) (c)

图 15-12 推箱子游戏的运行效果

参 考 文 献

［1］ Java 8 API 文档．http://docs.oracle.com/javase/8/docs/api/．

［2］ JDK 9 版文档．http://openjdk.java.net/projects/jdk9/．

［3］ 王克宏．Java 技术教程(基础篇)．北京：清华大学出版社，2005．

［4］ 耿祥义．Java 课程设计．2 版．北京：清华大学出版社，2008．

［5］ 朱战立，沈伟．Java 程序设计实用教程．北京：电子工业出版社，2005．

［6］ IVOR HORTON．Java 2 入门经典．叶乃文，等译．北京：机械工业出版社，2001．

［7］ 雍俊海．Java 程序设计教程．3 版．北京：清华大学出版社，2014．

［8］ 李芝兴，杨瑞龙．Java 程序设计之网络编程．2 版．北京：清华大学出版社，2009．

［9］ Bruce Eckel．Java 编程思想．陈昊鹏，译．北京：机械工业出版社，2007．

［10］ Peter val der Linden．Java 2 教程．刑国庆，译．北京：电子工业出版社，2005．

［11］ 陆迟．Java 语言程序设计．3 版．北京：电子工业出版社，2010．

［12］ 刘晓华等．JSP 应用开发详解．3 版．北京：电子工业出版社，2007．

［13］ 聂哲，袁梅冷，杨淑萍．Java 面向对象程序设计．北京：高等教育出版社，2005．

［14］ 高晓黎．Java 程序设计．2 版．北京：清华大学出版社，2015．

［15］ 李刚．疯狂 Java 讲义．3 版．北京：电子工业出版社，2014．

［16］ 明日科技．Java 从入门到精通(实例版)．北京：清华大学出版社，2012．



普通高等教育“十一五”国家级规划教材
21世纪大学本科计算机专业系列教材

近期出版书目

- 计算概论(第 2 版)
- 计算概论——程序设计阅读题解
- 计算机导论(第 3 版)
- 计算机导论教学指导与习题解答
- 计算机伦理学
- 程序设计导引及在线实践(第 2 版)
- 程序设计基础(第 2 版)
- 程序设计基础习题解析与实验指导(第 2 版)
- 程序设计基础(C 语言)(第 2 版)
- 程序设计基础(C 语言)实验指导(第 2 版)
- Java 程序设计(第 2 版)
- 离散数学(第 3 版)
- 离散数学习题解答与学习指导(第 3 版)
- 数据结构与算法
- 算法设计与分析(第 2 版)
- 算法设计与分析习题解答与学习指导(第 2 版)
- 数据结构(STL 框架)
- 形式语言与自动机理论(第 3 版)
- 形式语言与自动机理论教学参考书(第 3 版)
- 数字逻辑
- 计算机组成原理(第 3 版)
- 计算机组成原理教师用书(第 3 版)
- 计算机组成原理学习指导与习题解析(第 3 版)
- 微型计算机系统与接口(第 2 版)
- 计算机组成与系统结构(第 2 版)
- 计算机组成与体系结构习题解答与教学指导(第 2 版)
- 计算机组成与体系结构(第 2 版)
- 计算机系统结构教程
- 计算机系统结构学习指导与题解
- 计算机操作系统(第 2 版)
- 计算机操作系统学习指导与习题解答
- 数据库系统原理
- 编译原理
- 软件工程(第 2 版)
- 计算机图形学
- 计算机网络(第 4 版)
- 计算机网络教师用书(第 4 版)
- 计算机网络实验指导书(第 3 版)
- 计算机网络习题解析与同步练习
- 计算机网络软件编程指导书
- 人工智能
- 多媒体技术原理及应用(第 2 版)
- 算法设计与分析(第 3 版)
- 算法设计与分析习题解答(第 3 版)
- C ++ 程序设计(第 3 版)
- 面向对象程序设计(第 3 版)
- 计算机网络工程(第 2 版)
- 计算机网络工程实验教程
- 信息安全原理及应用